Intel Software

INTRODUCTION TO INTEL® VTUNE™ PROFILER & INTEL® ADVISOR

Matthew Cordery

Application Engineer

matthew.cordery@intel.com

Notices and Disclaimers

DISTRIBUTION STATEMENT: None Required

Disclosure Notice: This presentation is bound by Non-Disclosure Agreements between Intel Corporation and the Department of Energy, and Argonne National Lab, and is therefore for Internal Use Only and not for distribution outside these organizations or publication outside the above referenced Subcontracts.

Intel Corp Proprietary Information: This document contains trade secrets and/or proprietary information of Intel Corporation and Intel Federal LLC ("Intel") and is exempt from disclosure under the Freedom of Information Act. The information contained herein shall not be duplicated, used or disclosed outside the U.S. Department of Energy, UChicago Argonne LLC except as permitted by the contract previously referenced. The data subject to this restriction are contained in all sheets of this document.

USG Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Export Control: This document contains information that is subject to export control under the Export Administration Regulations. However the contents remain within the applicable ECCN's provided in the most recent Multi Party for Intel Restricted Secret Information that is applicable to the CORAL Aurora Program.

Intel Disclaimer: Intel makes available this document and the information contained herein in furtherance of the CORAL Aurora Program. None of the information contained herein is, or should be construed, as advice. While Intel makes every effort to present accurate and reliable information, Intel does not guarantee the accuracy, completeness, efficacy, or timeliness of such information. Use of such information is voluntary, and reliance on it should only be undertaken after an independent review by qualified experts.

Access to this document is with the understanding that Intel is not engaged in rendering advice or other professional services. Information in this document may be changed or updated without notice by Intel.

This document contains copyright information, the terms of which must be observed and followed.

Reference herein to any specific commercial product, process or service does not constitute or imply endorsement, recommendation, or favoring by Intel or the US Government.

Intel makes no representations whatsoever about this document or the information contained herein. IN NO EVENT SHALL INTEL BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, OR OTHERWISE, EVEN IF INTEL IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Get the tools

- Advisor and VTune are now part of the Intel® oneAPI Base Toolkit
  - Download entire toolkit
  - Download just VTune and Advisor (customizable installation)
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html
  - Available for Windows, Linux, MacOS (view Linux results)

Agenda

- Advisor – Intel's vectorization and optimization tool
  - CPU Optimization
    - Roofline
  - GPU Offloading
    - Offload Advisor
    - Roofline
- VTune – Intel's performance metric investigation tool
  - CPU capabilities
  - GPU metrics

Tuning at Multiple Hardware Levels

Exploiting all features of modern processors requires good use of the available resources

- Core
  - Vectorization is critical with 512bit FMA vector units (32 DP ops/cycle)
  - Cache use needed to feed vector units
- Socket
  - Using all cores in a processor requires parallelization (MPI*, OMP*,CUDA*,OPENCL*,SYCL*,DPC++ …)
  - Using coherent, shared socket caches
- Node
  - Minimize remote memory access (control memory affinity)
  - Minimize resource sharing (tune local memory access, disk IO and network traffic)

ADVISOR: NBODY DEMONSTRATION

The naïve code that could

N-body code

- Dr. Fabio Baruffa (original): https://github.com/fbaru-dev/nbody-demo
- Paulius Velesko (includes gpu): https://github.com/pvelesko/nbody-demo.git
  - Basically, the code in this demo

Nbody gravity simulation

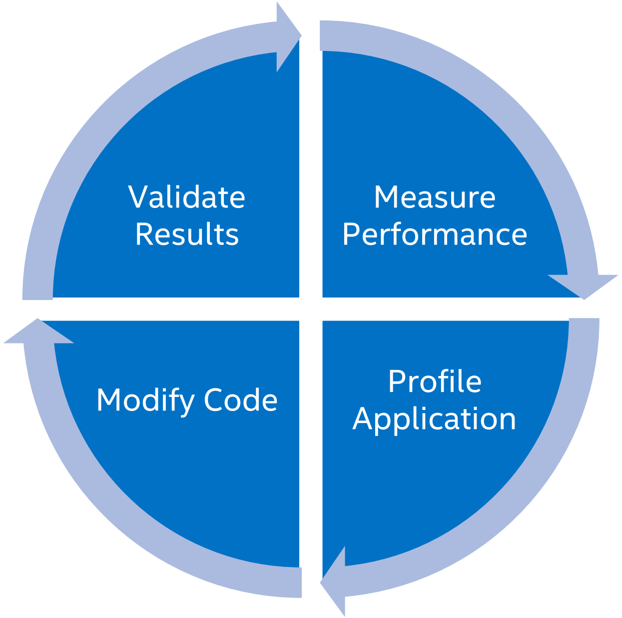
Consider a distribution of n point masses located at r_i with masses m_i and velocities and accelerations v_i and a_i, respectively

We want to calculate the position of the particles after a certain time interval using Newton's law of gravity.

```
struct Particle
{
 public:
   Particle() { init();}
   void init()
   {
     pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
     vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
     acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
     mass   = 0.;
   }
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
for (i = 0; i < n; i++){         // update acceleration
   for (j = 0; j < n; j++){
     real_type distance, dx, dy, dz;
     real_type distanceSqr = 0.0;
     real_type distanceInv = 0.0;

     dx = particles[j].pos[0] - particles[i].pos[0];
    …

     distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
    distanceInv = 1.0 / sqrt(distanceSqr);

     particles[i].acc[0] += dx * G * particles[j].mass *
                            distanceInv * distanceInv * distanceInv;
     particles[i].acc[1] += …
     particles[i].acc[2] += …
```

The Basic Tuning Cycle



Infinite cycle only broken by external constraints (time, papers, releases …)

Procedures for measuring performance and validating results are critical

Automation and **environment** control are key for **consistency**

Where do I start?

Version Optimizations

- Ver0
  - Initial implementation
- Ver1
  - Vectorized with compiler flags (march/mtune)
- Ver2
  - Use only floats
- Ver3/4
  - AoS -> SoA + SIMD Reduce
- Ver 7
  - OpenMP with data alignment

INTEL® ADVISOR

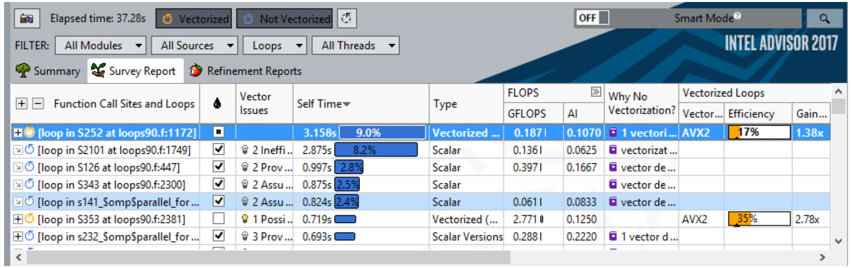
Vectorization and Static Analysis

https://www.alcf.anl.gov/user-guides/advixe-cl-xc40

Intel® Advisor – Vectorization Optimization

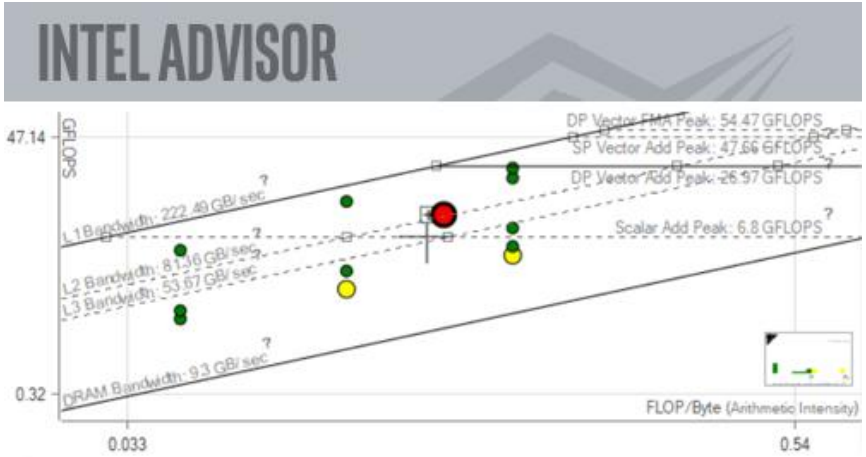
Faster Vectorization Optimization:

- Vectorize where it will pay off most
- Quickly ID what is blocking vectorization
- Tips for effective vectorization
- Safely force compiler vectorization
- Optimize memory stride



Roofline model analysis:

- Automatically generate roofline model
- Evaluate current performance
- Identify boundedness



Add Parallelism with Less Effort, Less Risk and More Impact

http://intel.ly/advixe-cl-xe

Typical Vectorization Optimization Workflow

There is no need to recompile or relink the application, but the use of **-g** is recommended.

Note: if you're using Theta run out of /projects rather than /home

- Collect survey (overhead ~5%) advixe-cl -c survey
  - Basic info (static analysis) - ISA, time spent, etc.
- Collect roofline advixe-cl -c roofline
  - Basically the survey analysis above with roofline analysis (trip counts, flops)
- Collect dependencies (overhead 5-1000x) advixe-cl -c dependencies
  - Differentiate between real and assumed issues blocking vectorization
- Collect Memory Access Patterns advixe-cl -c map
  - Get advice on memory strides
- NB: You can run multiple analyses, and sometimes you have to, to get all of the information you need.
  - For example, in the same batch job you can do a roofline and a dependency analysis and have the output directory be the same so all of that information is shown together in a single context.
  - Make sure you create different output directories for different experiments.

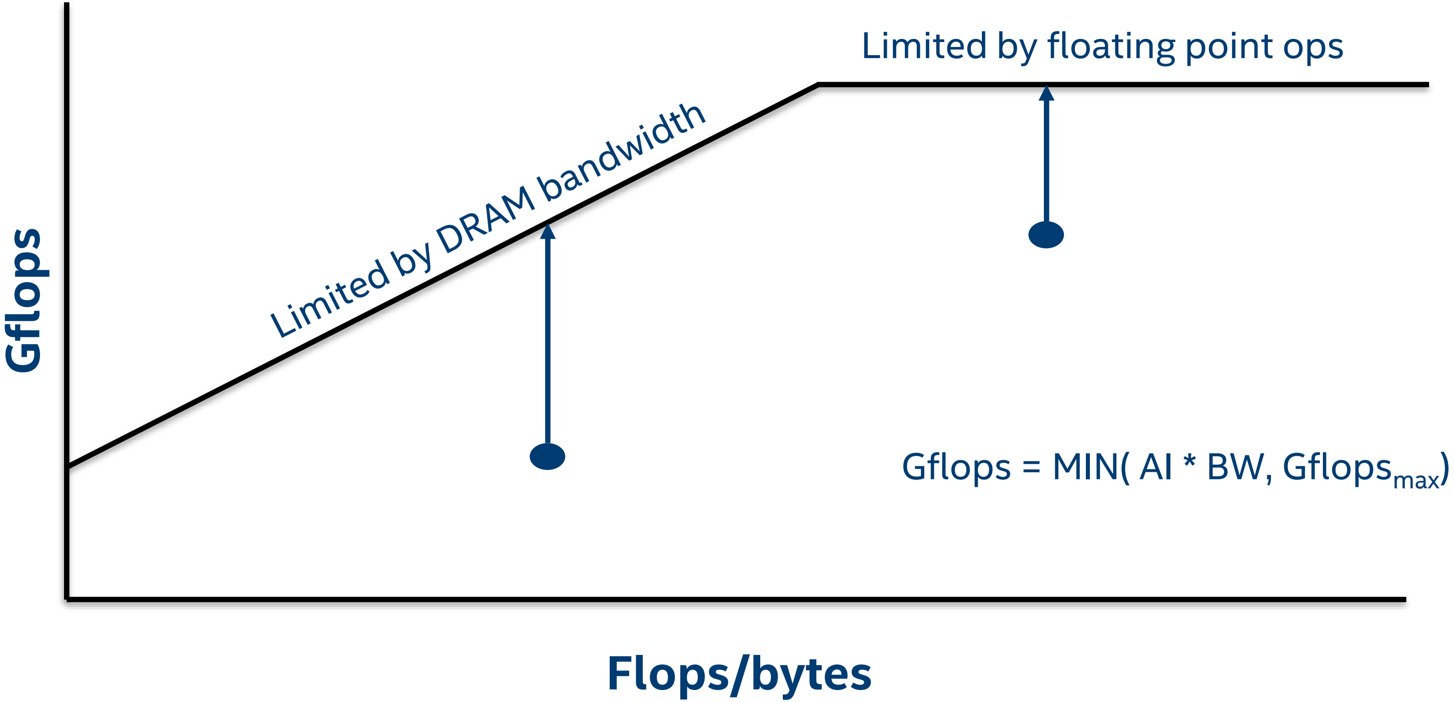
What is a roofline?

- A roofline is a graphical representation of two factors that affect code performance: flops & memory bandwidth
    - Codes at scale may be limited by file I/O or MPI but there can be different rooflines for those cases.
- Allows you answer questions like:
    - What is/isn't limiting this kernel's performance? Which kernels are more important to overall code performance?
    - What gains might I see from focusing on a particular kernel?
    - Where do I need to focus my software engineering efforts to achieve further gains?

Roofline cont'd

- Arithmetic intensity (AI)
  - Flops / bytes ratio
    - Bytes can be data moved to/from DRAM, cache, etc.
    - Kernels with a high AI are limited by chip floating point performance (e.g. DGEMM)
    - Kernels with a low AI are limited by memory bandwidth (e.g. STREAM triad, many HPC computational physics kernels)
    - Kernels can be limited by both

Roofline



Limited by floating point ops
Limited by DRAM bandwidth
Gflops
Gflops = MIN(AI * BW, Gflops_max)
Flops/bytes

Cache-Aware Roofline Optimization

Next Steps

If Under the Vector Add Peak

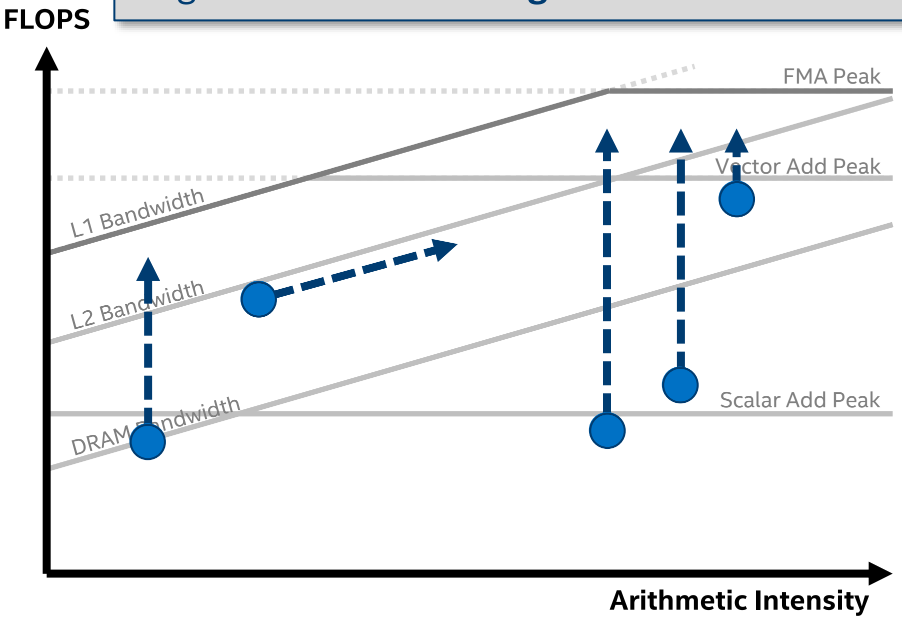
Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage.**

If just above the Scalar Add Peak

Check **vectorization efficiency** in the Survey. Follow the recommendations to improve it if it’s low.

If under or near a memory roof…

- Try a MAP analysis. Make any appropriate **cache optimizations.**
- If cache optimization is impossible, try **reworking the algorithm to have a higher AI.**



If under the Scalar Add Peak…

Check the Survey Report to see if the loop vectorized. If not, try to **get it to vectorize** if possible. This may involve running Dependencies to see if it’s safe to force it.

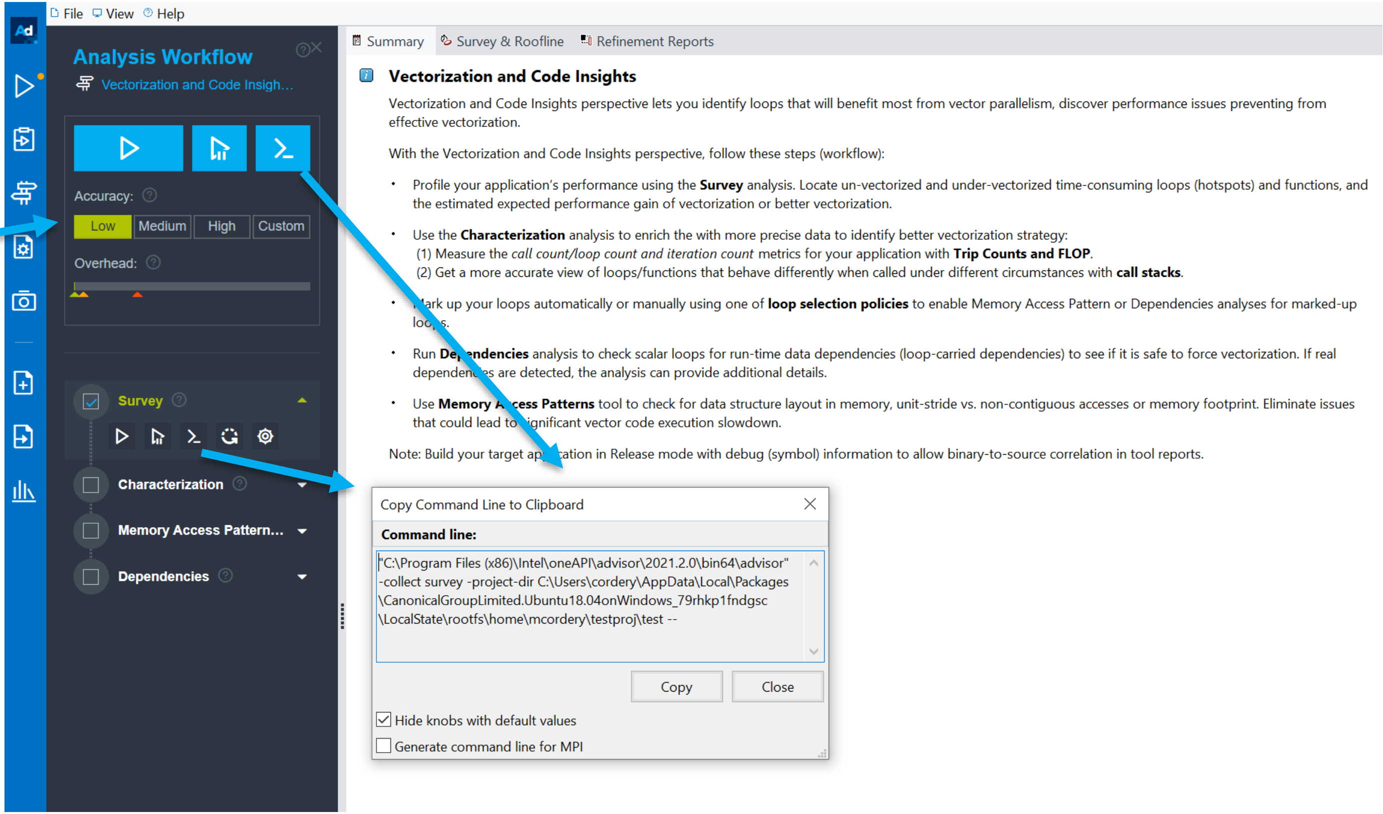
Use --help option!

advixe-cl --help collect

```
Examples:

 Perform a Survey analysis to determine hotspots.

        advisor --collect=survey --project-dir=./advi --search-dir src:r=./src
                -- ./bin/myApplication

 Perform a Memory Access Patterns analysis on the specified loops.

        advisor --collect=map --mark-up-list=5,10,12 --project-dir=./advi --search-dir src:r=./src
                -- ./bin/myApplication

 Perform a Survey analysis on four nodes of the MPI cluster and store the collected data in the shared ./advi project directory.

        mpirun -n 4 advisor --project-dir=./advi --collect=survey
                -- <PATH>/mpi-sample/1_mpi_sample_serial

 Perform a Dependencies analysis on all innermost loops that run above 2% of the total CPU time.

        advisor --collect=dependencies --project-dir=./advi --loops="loop-height=0,total-time>2"
                 -- ./bin/myApplication

 Perform a Roofline analysis.

        advisor --collect=roofline --project-dir=./advi -- ./bin/myApplication
```

Generate Advisor Command Lines from the GUI

How accurate you want your reports to be



Collect survey and tripcounts (roofline)

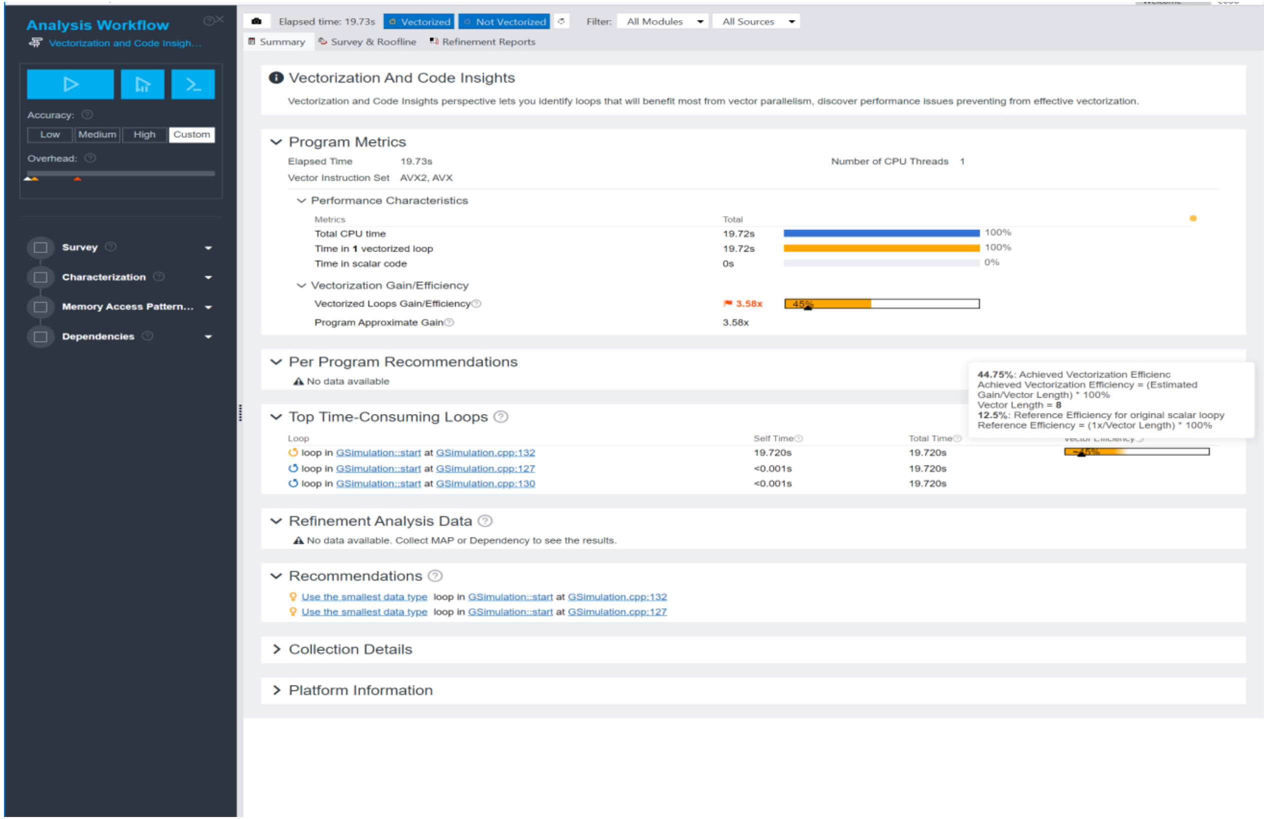
```
$ Theta: module load PrgEnv-intel amplxe-cl advixe-cl

$ cd project; make

#!/bin/bash

..stuff..

advixe-cl –collect=roofline –trip-counts –project-dir=<project-dir> -- <executable> <parameters>
```

Where your results go

View Result on Local Machine

- Make sure your local version of Advisor (or VTune) is at least the same as that of the one used to generate the data otherwise errors might occur.
- X-forwarding is not recommended.
- Tar the result along with sources and binary (if you want to be able to view them, unless you already have them locally)
- Copy to your local machine
- May have to point advixe-cl at your local sources and binary

Summary Report

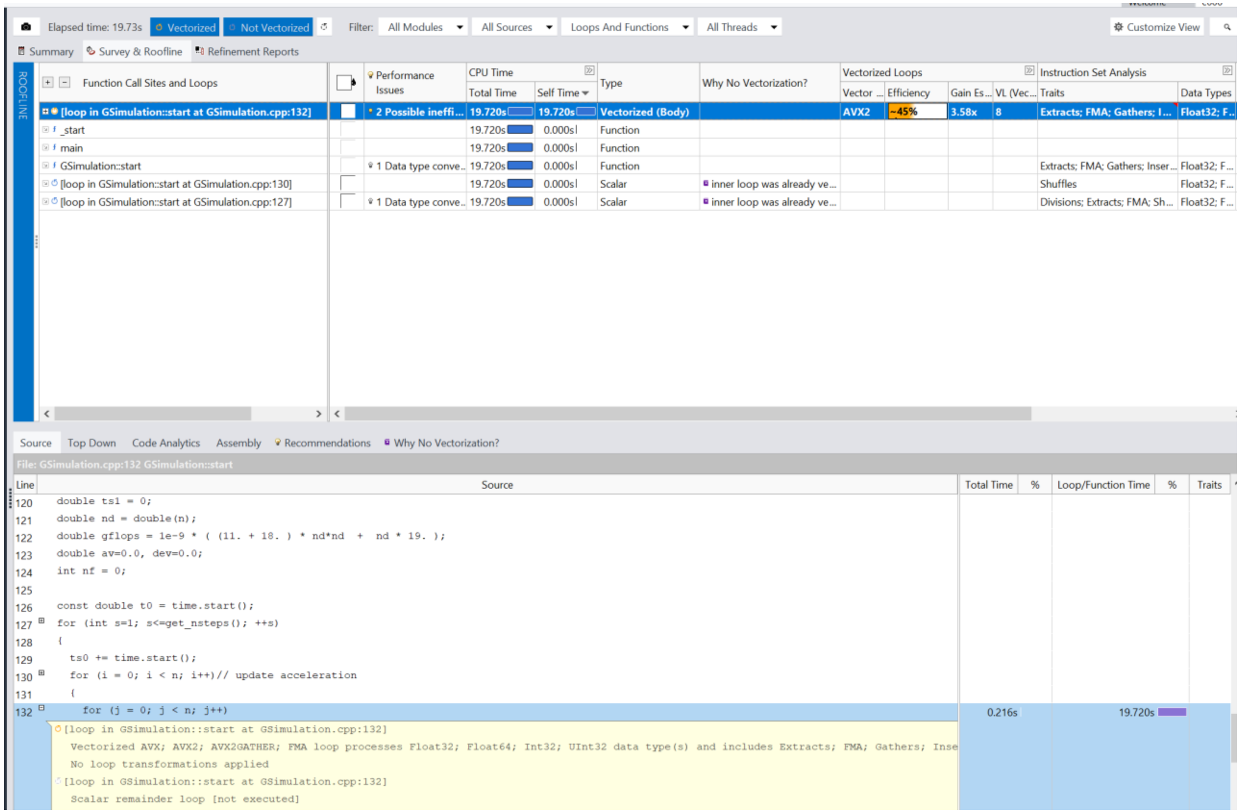


Summary provides overall performance characteristics

Top time consuming loops are listed individually

Vectorization efficiency is based on used ISA

Survey Report (Source Tab)

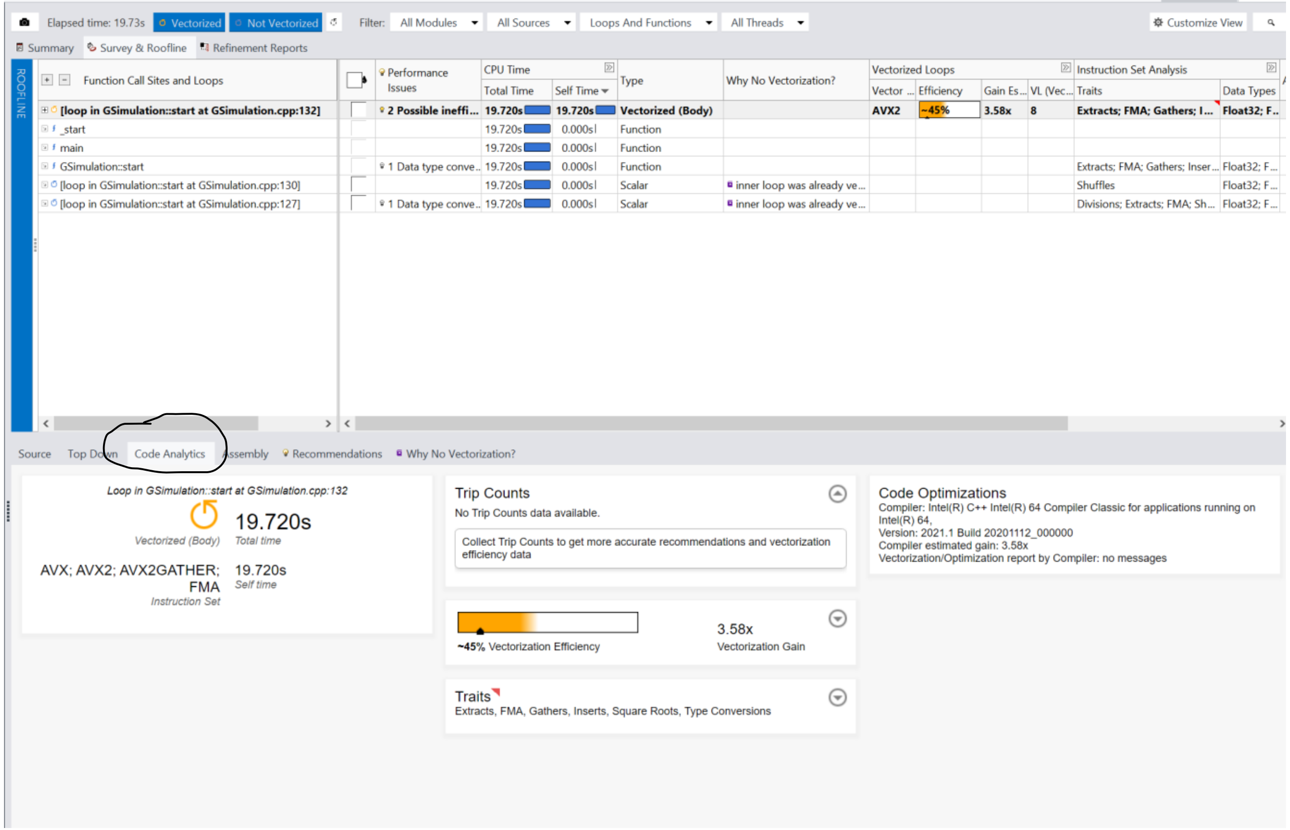


Notice the following:

- Vector ISA
- Type Conversions
- Memory Access Patterns

All of these elements may affect performance

Survey Report (Code Analytics Tab)



Analytics tab contains a wealth of information

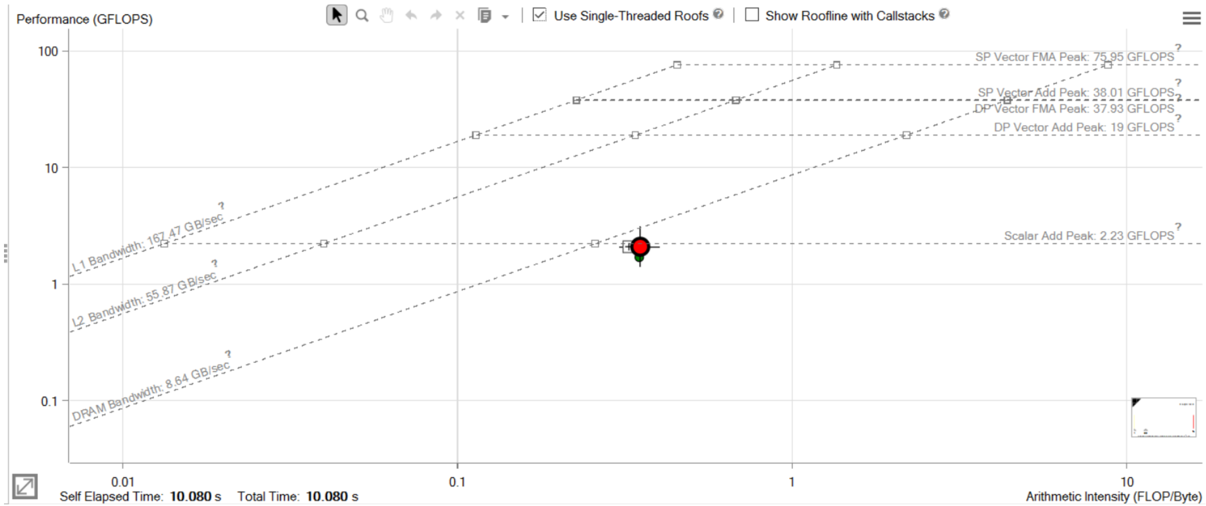
- Instruction set
- Instruction mix
- Traits (sqrt, type conversions, unpacks)
- Vector efficiency
- Floating point statistics

And explanations on how they are measured or calculated – expand the box or hover over the question marks.

LIVE DEMO

Roofline

CARM (Cache-aware roofline model) Analysis



Using single threaded roof

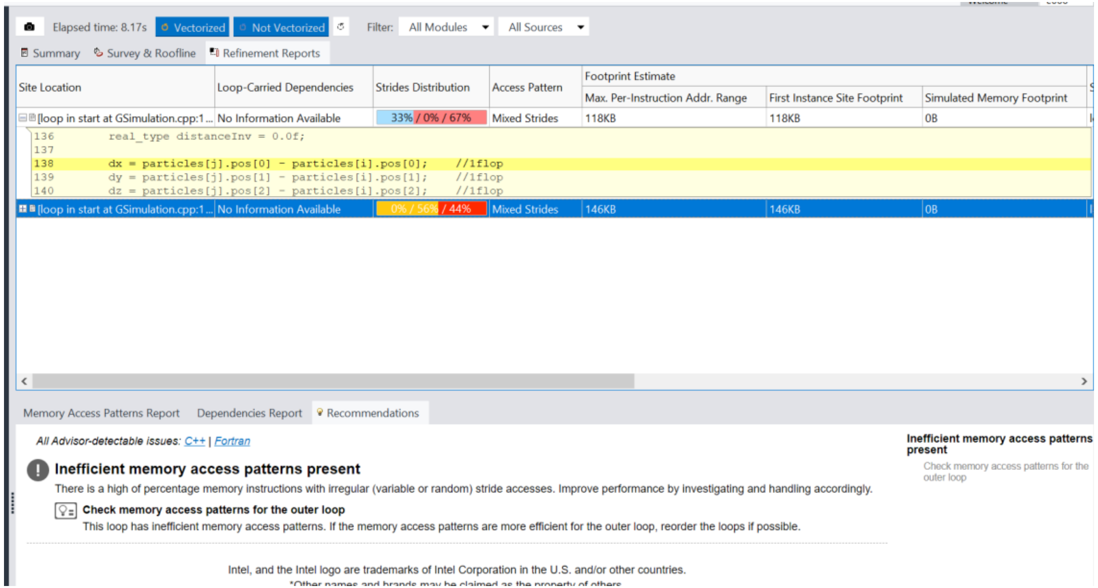
Code vectorized, but performance on par with scalar add peak?

- Irregular memory access patterns force gather operations.
- Overhead of setting up vector operations reduces efficiency.

Next step is clear: perform a Memory Access Pattern analysis

Memory Access Pattern Analysis (Refinement)

```
advixe-cl –c roofline –r mydat ./nody.x 4000 500
advixe-cl –c map –r mydat ./nbody.x 4000 500
```



Storage of particles is in an Array Of Structures (AOS) style

This leads to regular, but non-unit strides in memory access

- 33% unit
- 0% uniform, non-unit
- 67% non-uniform

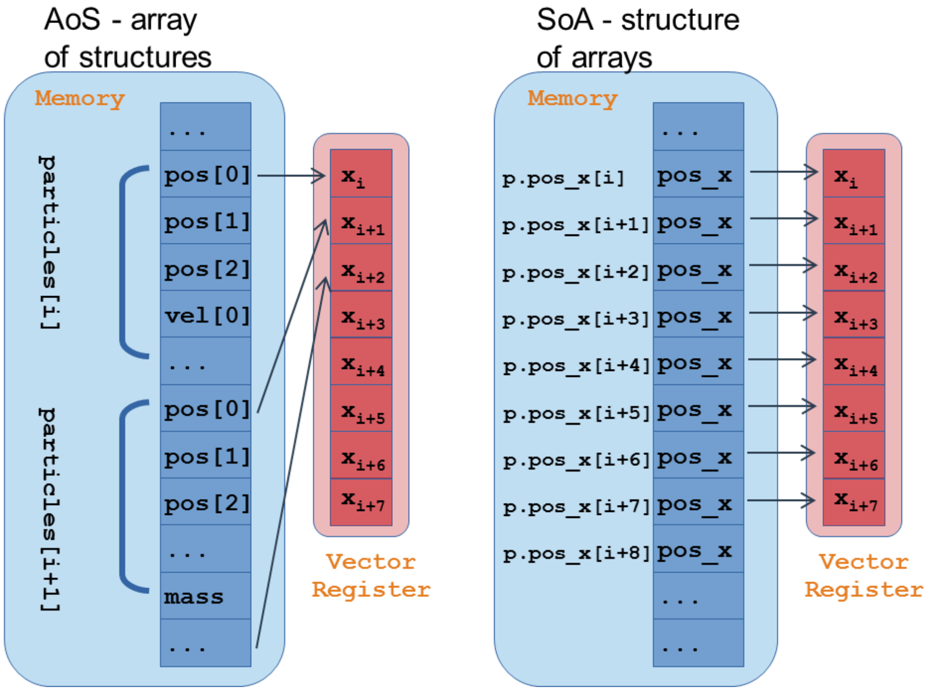
Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization

Vectorization: gather/scatter operation

The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous

```
struct Particle
{
 public:
   ...
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
struct ParticleSoA
{
 public:
   ...
   real_type *pos_x,*pos_y,*pos_z;
   real_type *vel_x,*vel_y,*vel_z;
   real_type *acc_x,*acc_y;*acc_z
   real_type *mass;
};
```



Memory access pattern analysis

How should I access data ?

Best: Unit stride access are faster

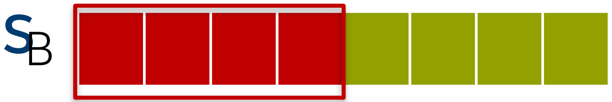
```
for (i=0; i<N; i++)
    A[i] = B[i]*d
```

OK: Constant stride are more complex

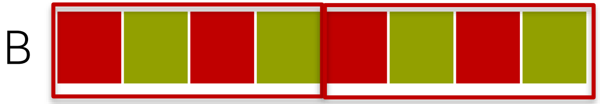
```
for (i=0; i<N; i+=2)
    A[i] = B[i]*d
```

Bad: Irregular access

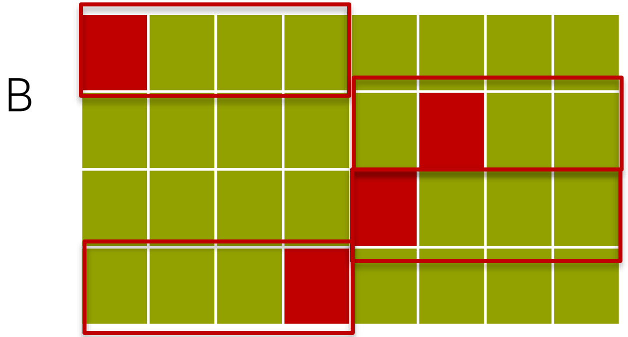
```
for (i=0; i<N; i++)
    A[i] = B[C[i]]*d
```



For B, 1 cache line load computes 4 DP



For B, 2 cache line loads compute 4 DP with reconstructions



For B, 4 cache line loads compute 4 DP with reconstructions, prefetching might not work

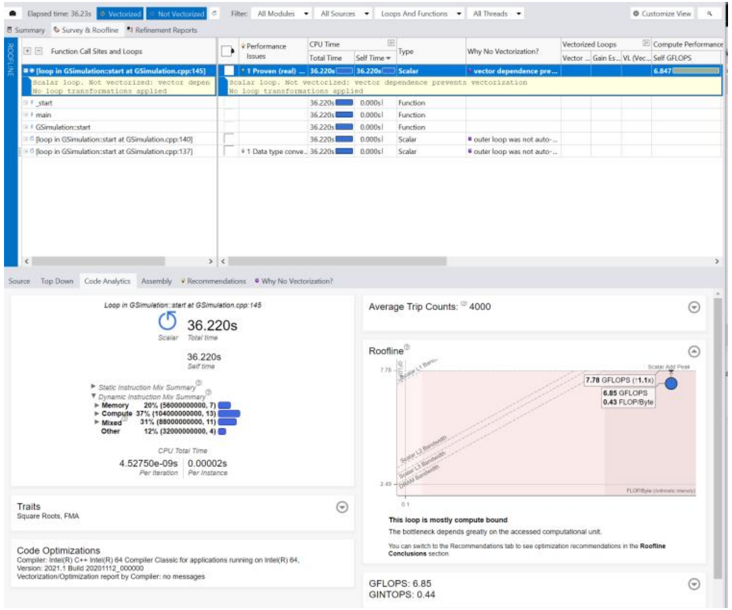
Performance After Data Structure Change

In this new version (version 3 in GitHub sample) we introduce the following change:

- Change particle data structures from AOS to SOA

Note changes in report:

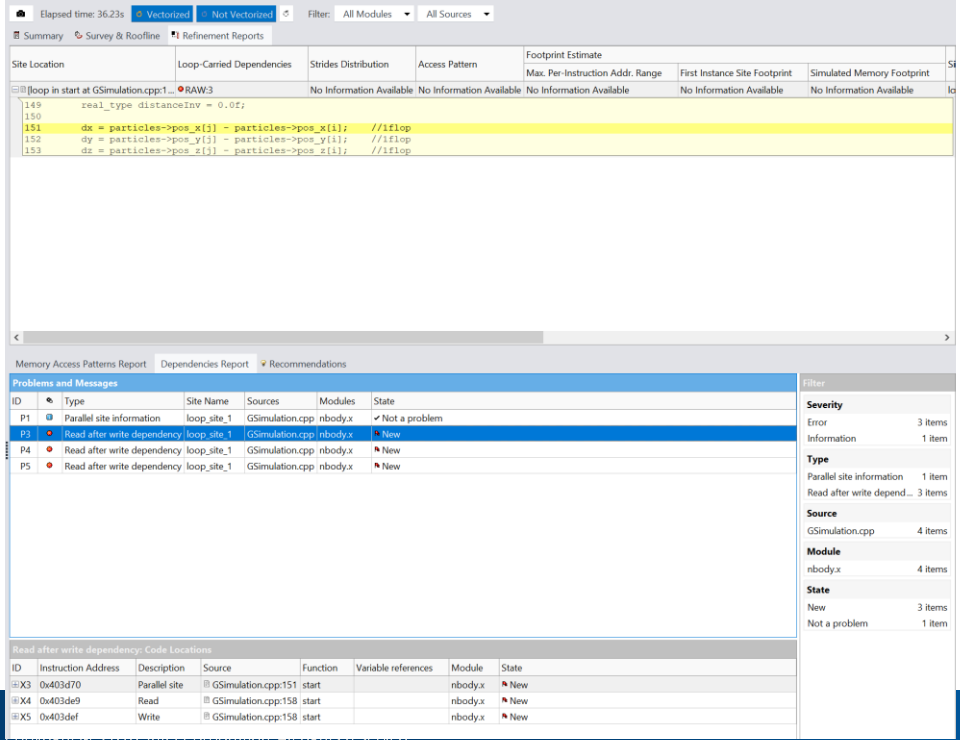
- Performance is lower
- Main loop is no longer vectorized
- Assumed vector dependence prevents automatic vectorization



Next step is clear: perform a Dependencies analysis

Dependencies Analysis (Refinement)

```
Run "survey" followed by "dependencies"
advixe-cl –c dependencies ./nbody.x 4000 500
```



Dependencies analysis has high overhead:

- Run on reduced workload

Advisor Findings:

- RAW dependency

Recommendations

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

All Advisor-detectable issues: C++ | Fortran

Proven (real) dependency present

The compiler assumed there is an anti-dependency (Write after read - WAR) or true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Resolve dependency

The Dependencies analysis shows there is a real (proven) dependency in the loop. To fix: Do one of the following:

- If there is an anti-dependency, enable vectorization using the directive `#pragma omp simd safelen(length)` , where `length` is smaller than the distance between dependent iterations in anti-dependency.

  Example

  ```
  #pragma omp simd safelen(4)
  ...
  ```

- If there is a reduction pattern dependency in the loop, enable vectorization using the directive `#pragma omp simd reduction(operator:list)` .

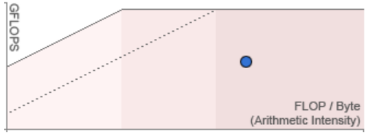
  Example

  ```
  #pragma omp simd reduction(+:sumx)
  ...
  ```

- Rewrite the code to remove the dependency. Use programming techniques such as variable privatization.

Roofline conclusions

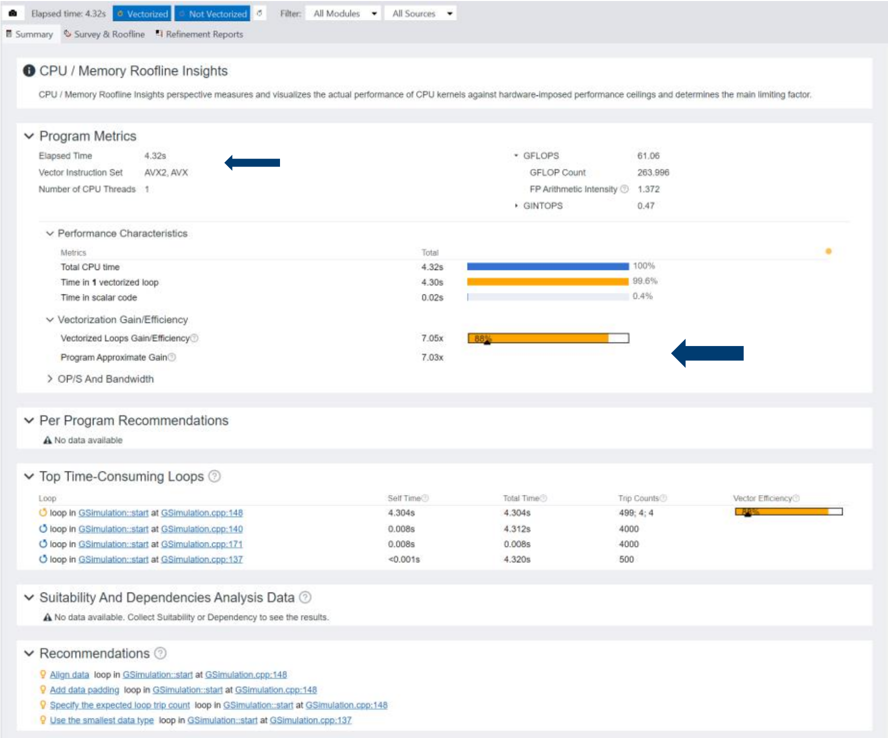
Conclusions, with optimization recommendations, are sorted by relevance.



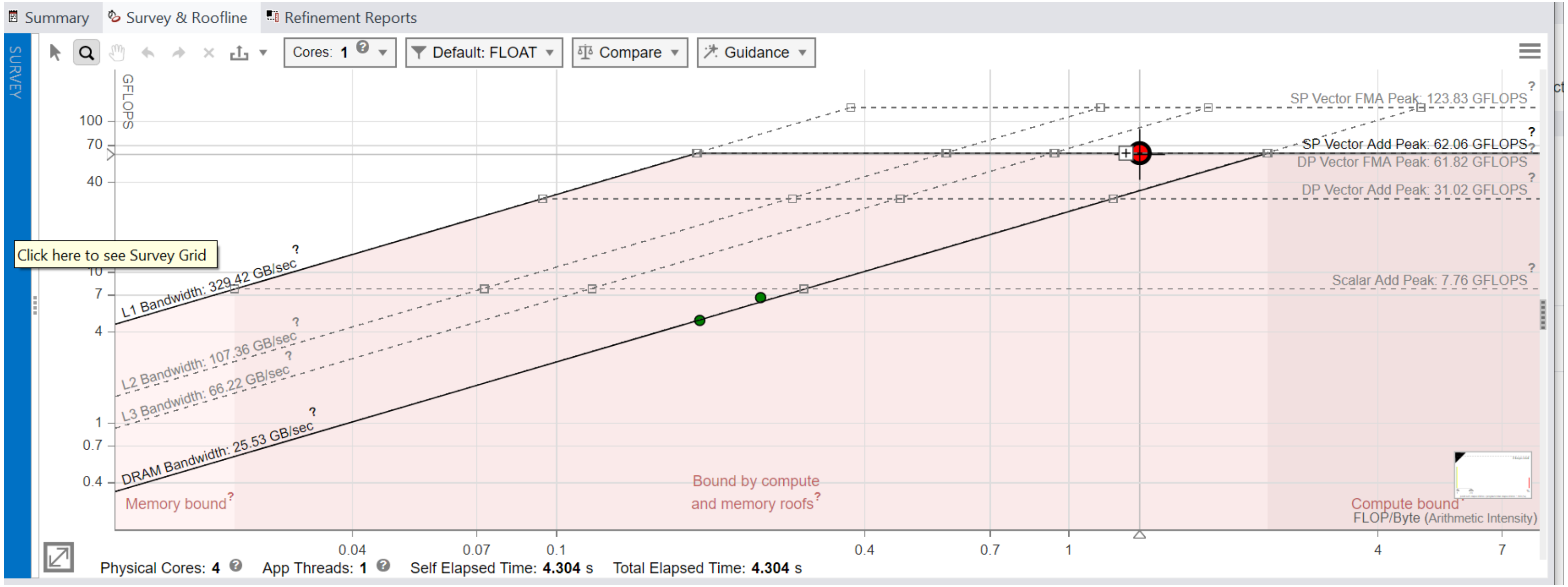
This loop is mostly compute bound

The bottleneck depends greatly on the accessed computational unit.
The loop is scalar. To fix: Vectorize the loop.

Performance after resolve dependencies

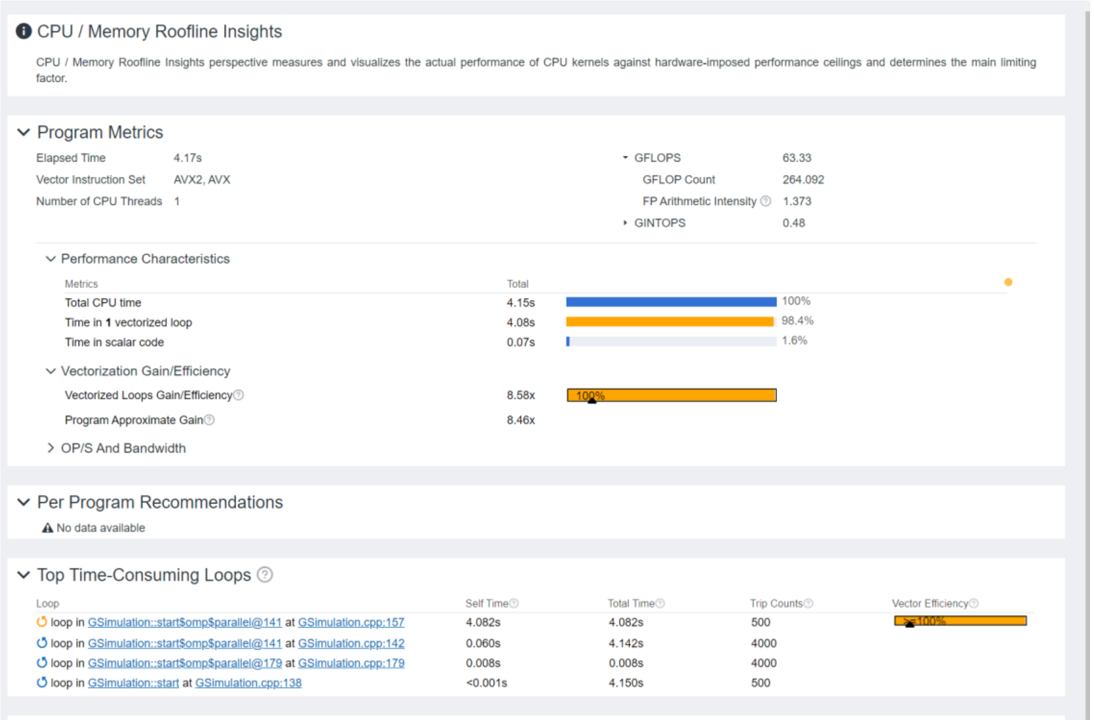


Performance After Resolved Dependencies



New memory access pattern plus vectorization produces much improved performance!
What's next? Try suggestions for aligning data.

Final performance



- Some additional performance eked out.
- Vectorization of loop now 100%
- At this point, you'll likely need to switch to VTune to begin investigating cache misses.

ADVISOR: GPU OFFLOAD

Which codes to migrate to GPU?

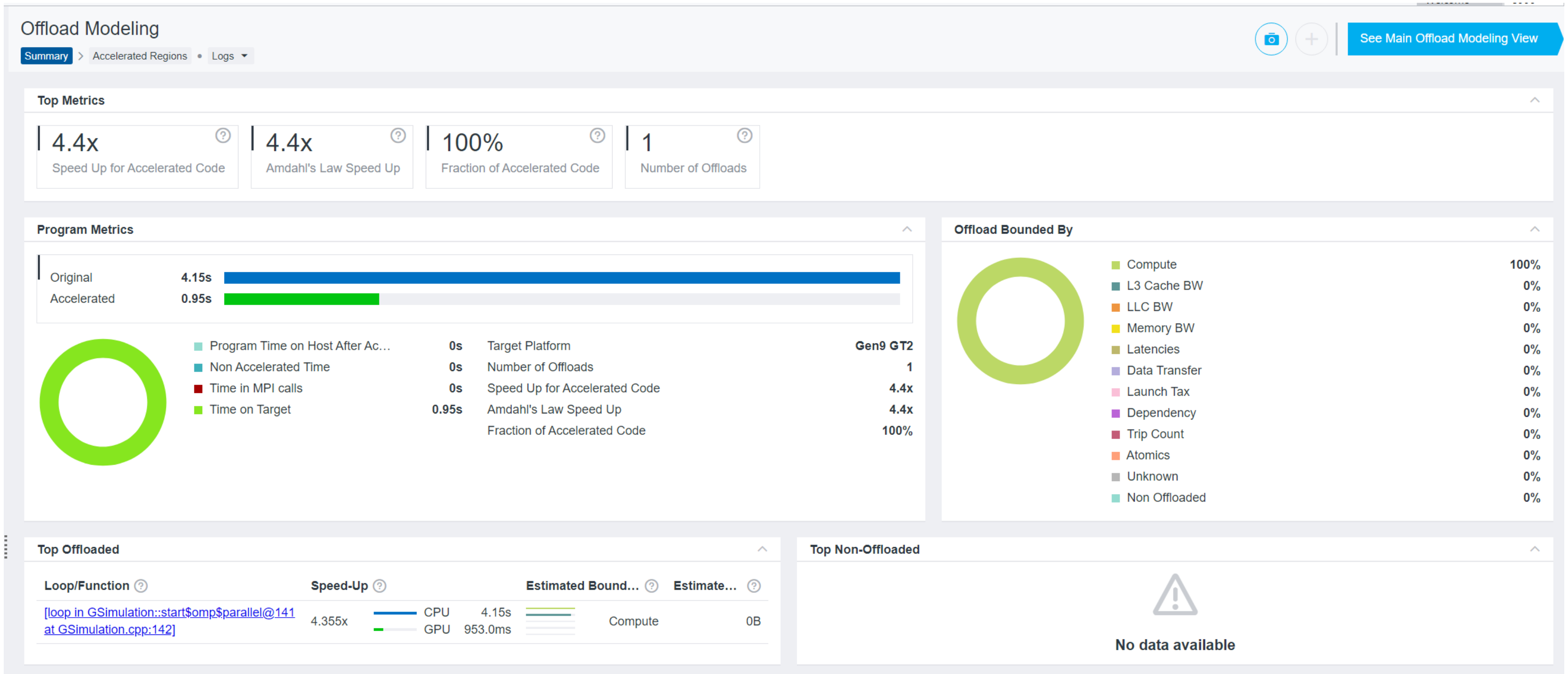
Offload Advisor

- Another option for accelerating loops is offloading them to an accelerator such as a GPU.
  - As with vectorization, Advisor now has the capability of allowing the user to test if kernels would benefit from offloading
    - Run a number of Advisor collections to generate data
    - Run a projection to a specific architecture (Intel only)
      - Report shows which loops would benefit from offloading, and which would not.

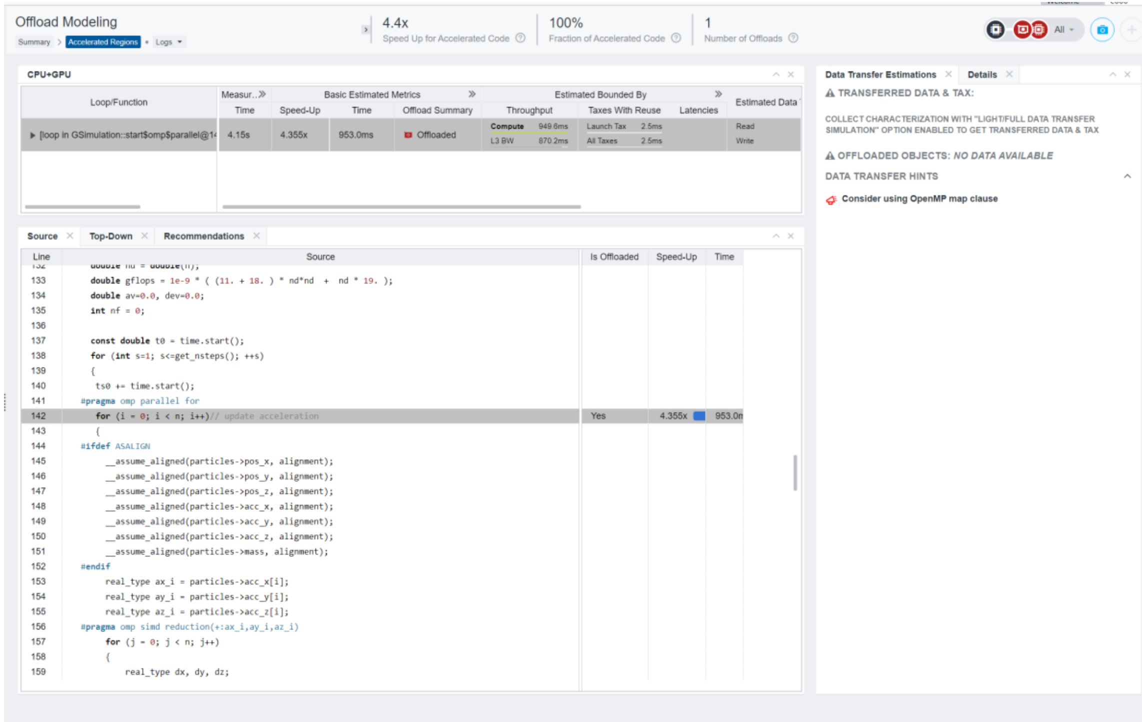
Nbody test case

- Take final optimized test case (ver7) with the main computational loop parallelized with OpenMP on the host (OpenMP not a requirement)
  - Collect a survey
    - `advixe-cl --collect=survey --project-dir=./advi_proj_v7 --stackwalk-mode=online --static-instruction-mix -- ./nbody.x 4000 500`
  - Collect flops and counts and target a particular device
    - `advixe-cl --collect=tripcounts --project-dir=./advi_proj_v7 --flop --target-device=gen9_gt2 -- ./nbody.x 4000 500`
  - Do a projection, targeting the same device
    - `advixe-cl --collect=projection  --project-dir=./advi_proj_v7 --`**`config=gen9_gt2`**` --no-assume-dependencies`

Offload summary

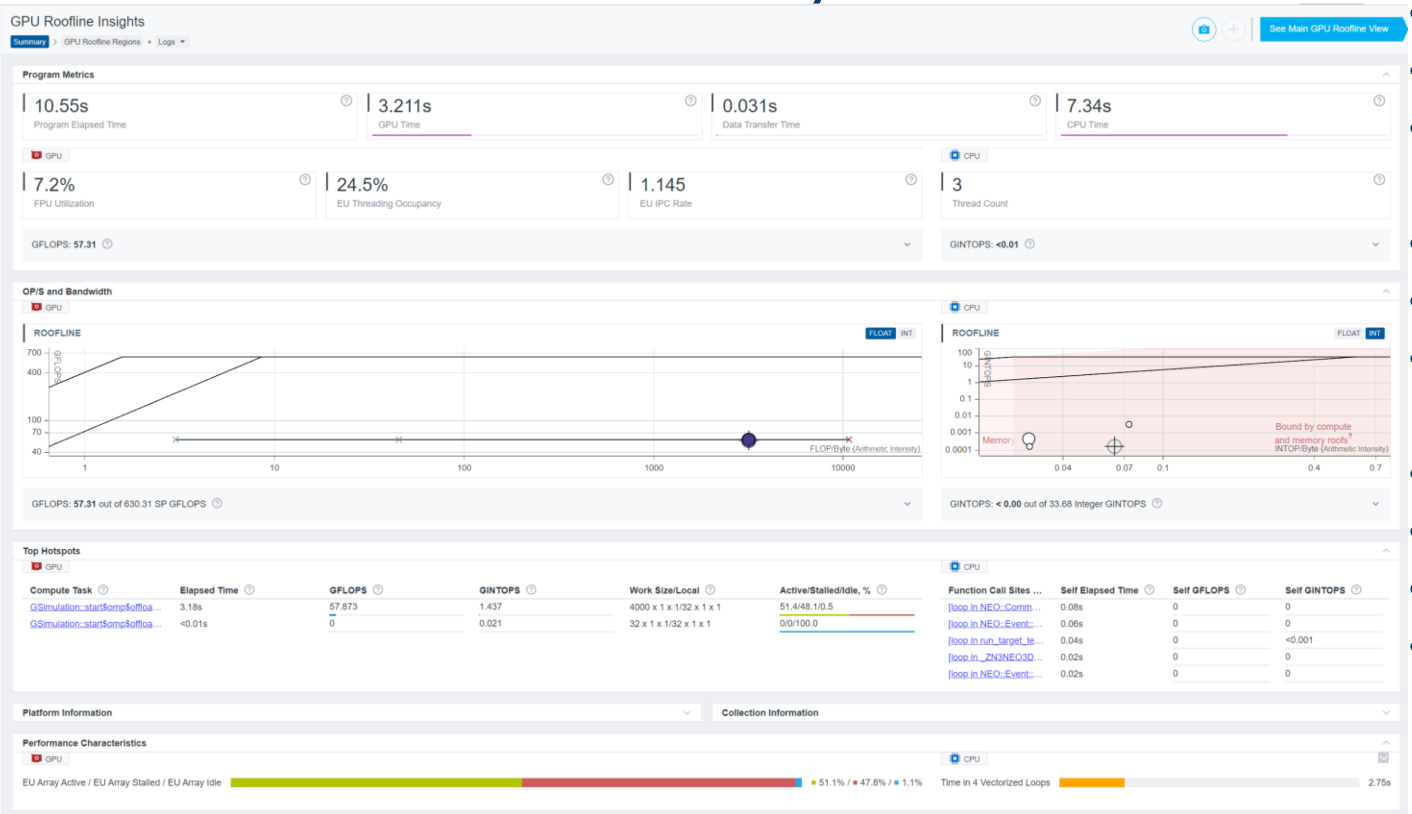


Offload modeling Accelerated regions tab



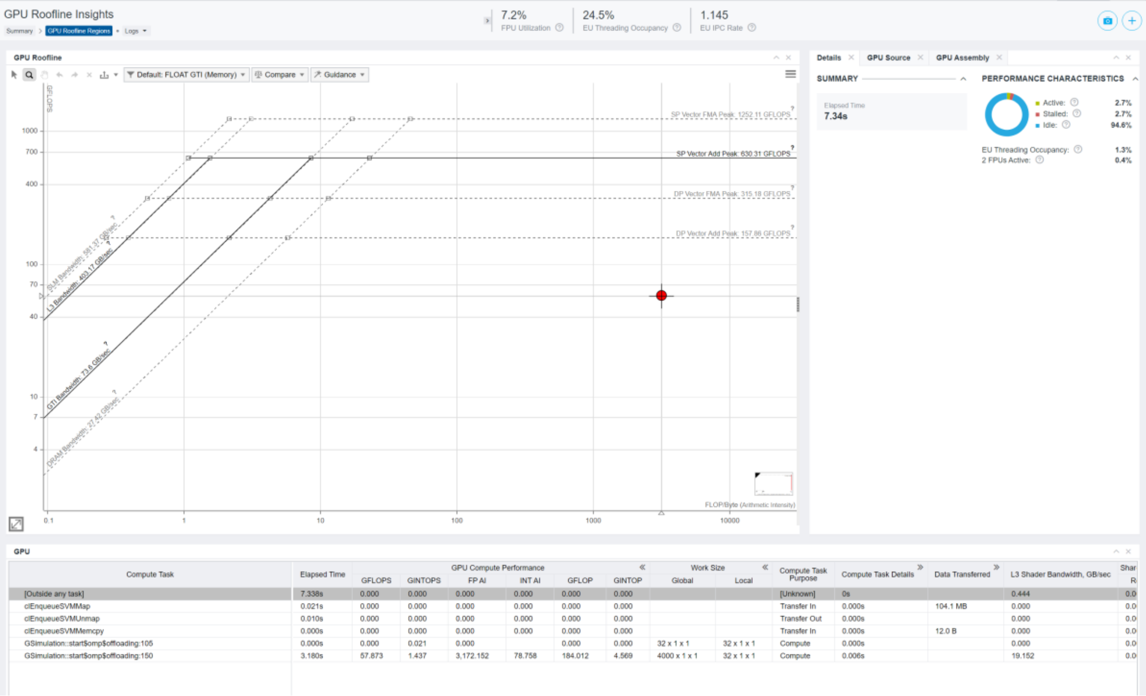
- Drill down on offloaded loop
  - Estimated speedup
  - Launch and data transfer latencies
- Offload loop with
  - OpenMP target directives and data mapping clauses
- Profile again with
  - `advixe-cl -collect=roofline -profile-gpu`

GPU Roofline Summary



- Program time
- GPU time
- Data transfer time
- CPU time
- FPU Utilization
- EU threading Occupancy
- IPC rate
- Thread count
- Roofline
- Hotspots

GPU Roofline Insights



Compute Task	Elapsed Time	GPU Compute Performance: GFLOPS	GINTOPS	FP AI	INT AI	GFLOP	GINTOP	Work Size: Global	Local	Compute Task Purpose	Compute Task Details	Data Transferred	L3 Shader Bandwidth, GB/sec
[Outside any task]	7.338s	0.000	0.000	0.000	0.000	0.000	0.000			[Unknown]	0s		0.444
clEnqueueSVMMap	0.021s	0.000	0.000	0.000	0.000	0.000	0.000			Transfer In	0.000s	104.1 MB	0.000
clEnqueueSVMUnmap	0.010s	0.000	0.000	0.000	0.000	0.000	0.000			Transfer Out	0.000s		0.000
clEnqueueSVMMemcpy	0.000s	0.000	0.000	0.000	0.000	0.000	0.000			Transfer In	0.000s	12.0 B	0.000
GSimulation::startompoffloading:105	0.000s	0.000	0.021	0.000		0.000	0.000	32 x 1 x 1	32 x 1 x 1	Compute	0.000s		0.000
GSimulation::startompoffloading:150	3.180s	57.873	1.437	3,172.152	78.758	184.012	4.569	4000 x 1 x 1	32 x 1 x 1	Compute	0.006s		19.152

- Measured GPU roofline
  - L3, SLM, GTI and DRAM bandwidth
- Kernel location
- FPU Utilization
- EU Threading Occupancy
- IPC rate
- Active/stalled/idle %

Data collection for specific regions: ittnotify

```
#include <ittnotify.h>
Int main(int argc, char* argv[] )
{
// do work here
__itt_pause();
// do more work
__itt_resume()
// Interesting work here
__itt_pause()
// Do more uninteresting work
Return 0;
}
```

- If you just want to collect/examine data from specific regions in your code, you can use ittnotify interface.
  - Link in libittnotify.a
  - Can start program with `__itt_pause()` or launch with `advixe-cl -start-paused`.
  - Can also use in VTune
  - Also a Fortran interface
    - e.g. CALL ITT_PAUSE()

INTEL® VTUNE™

Core-level hardware metrics

https://www.alcf.anl.gov/user-guides/amplxe-cl-xc40

Intel® VTune™ Amplifier

VTune is a full system (node level) profiler

- Accurate
- Low overhead
- Comprehensive (CPU, GPU, microarchitecture, memory, IO, threading, …)
- Configurable interface with easily accessed help
- Direct access to source code and assembly

Analyzing execution behavior with shared resources is critical in achieving good performance on multicore and offload processing systems

VTune Phases

Collection

- Record data
- Occurs while target executable is running

Finalization

- Calculations based on recorded data
- Used in displays / views
- Occurs after collection or in GUI (deferred)

Reports

- Static report
- Various formats: text, HTML, XML, CSV

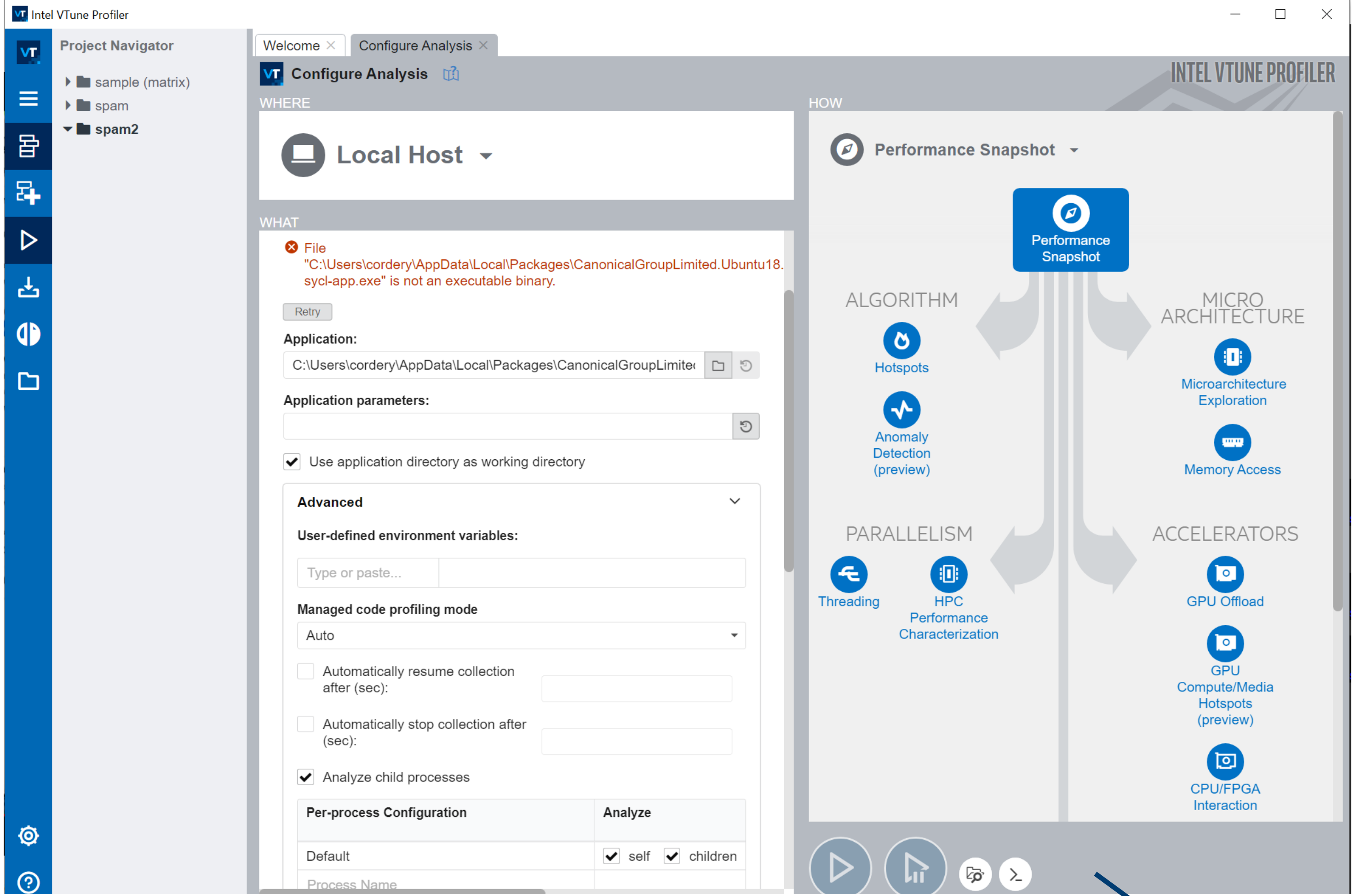
Display (Views)

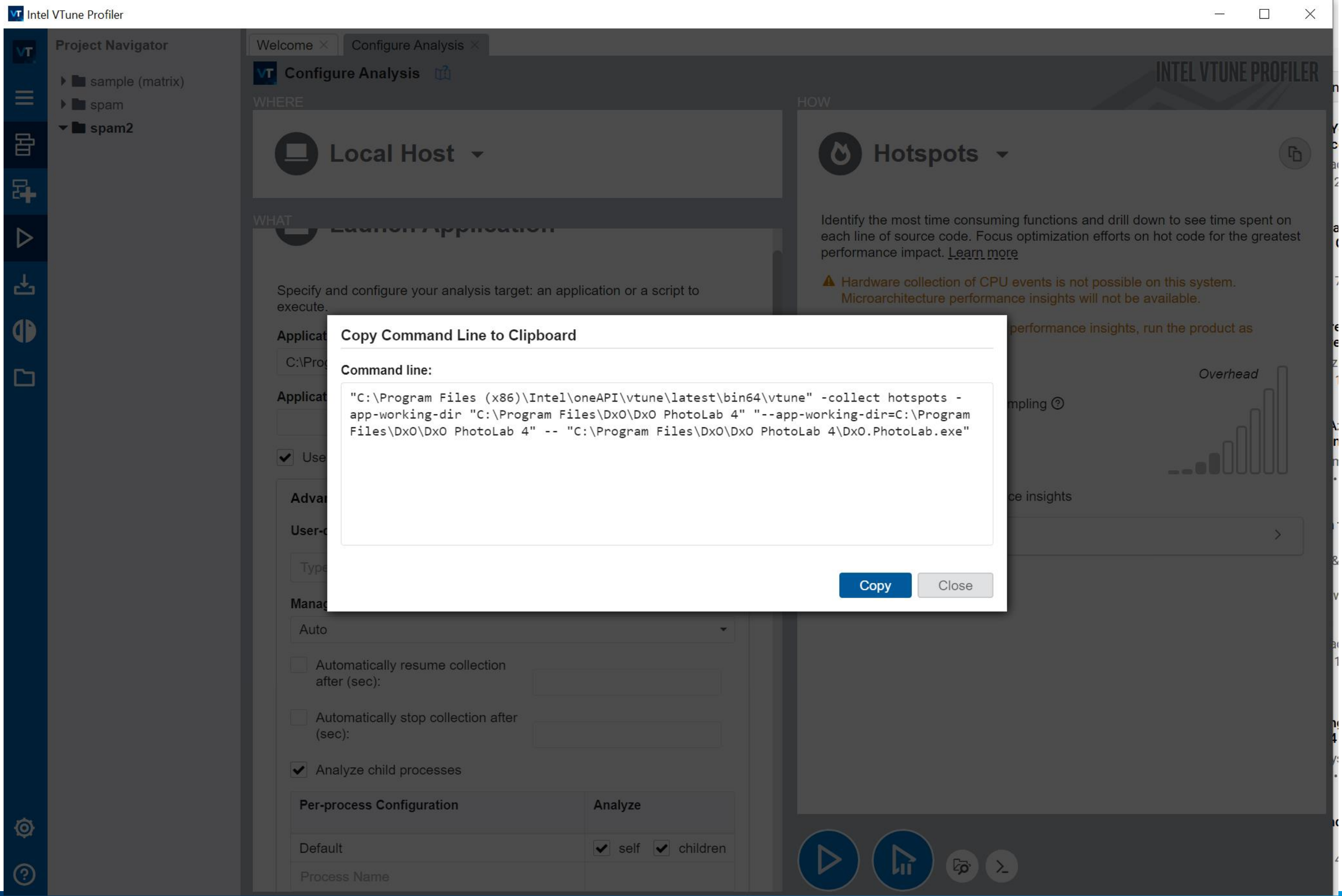
- VTune GUI
- Interactive, configurable

Predefined Collections

- Many available analysis types (only sme below):

• hotspots	Basic hotspots
• memory-consumption	Use of memory and allocation
• uarch-exploration	CPU microarchitecture bottlenecks
• memory-access	Memory access
• threading	Threading performance, overhead
• hpc-performance	OpenMP eff., memory access, vectorization,etc
• io	I/O subsystems, CPU, processor buses
• gpu-offload	Code execution on cpu and gpu
• gpu-hotspots	Hots spots, GPU hw metrics, mem latency, etc


Intel VTune Profiler
Project Navigator
sample (matrix)
spam
spam2
Welcome
Configure Analysis
Configure Analysis
INTEL VTUNE PROFILER
WHERE
Local Host
WHAT
File "C:\Users\cordery\AppData\Local\Packages\CanonicalGroupLimited.Ubuntu18. sycl-app.exe" is not an executable binary.
Retry
Application:
C:\Users\cordery\AppData\Local\Packages\CanonicalGroupLimite
Application parameters:
Use application directory as working directory
Advanced
User-defined environment variables:
Type or paste...
Managed code profiling mode
Auto
Automatically resume collection after (sec):
Automatically stop collection after (sec):
Analyze child processes
Per-process Configuration
Analyze
Default
self
children
Process Name
HOW
Performance Snapshot
Performance Snapshot
ALGORITHM
Hotspots
Anomaly Detection (preview)
MICRO ARCHITECTURE
Microarchitecture Exploration
Memory Access
PARALLELISM
Threading
HPC Performance Characterization
ACCELERATORS
GPU Offload
GPU Compute/Media Hotspots (preview)
CPU/FPGA Interaction


Intel VTune Profiler
Project Navigator
sample (matrix)
spam
spam2
Welcome
Configure Analysis
Configure Analysis
INTEL VTUNE PROFILER
WHERE
Local Host
HOW
Hotspots
WHAT
Specify and configure your analysis target: an application or a script to execute.
Identify the most time consuming functions and drill down to see time spent on each line of source code. Focus optimization efforts on hot code for the greatest performance impact. Learn more
Hardware collection of CPU events is not possible on this system. Microarchitecture performance insights will not be available.
Copy Command Line to Clipboard
Command line:
"C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune" -collect hotspots -app-working-dir "C:\Program Files\DxO\DxO PhotoLab 4" "--app-working-dir=C:\Program Files\DxO\DxO PhotoLab 4" -- "C:\Program Files\DxO\DxO PhotoLab 4\DxO.PhotoLab.exe"
Copy
Close
Overhead
Auto
Automatically resume collection after (sec):
Automatically stop collection after (sec):
Analyze child processes
Per-process Configuration
Analyze
Default
self
children
Process Name

HPC-Perf analysis: nbody demo (ver7: threaded)

HPC Performance Characterization HPC Performance Characterization

Analysis Configuration Collection Log Summary Bottom-up

Effective Physical Core Utilization: 96.7% (3.868 out of 4)

Vectorization: 100.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	62.3%	of uOps
Packed:	100.0%	from SP FP
Scalar:	0.0%	from SP FP
DP FLOPs:	0.0%	of uOps
Packed:	0.0%	from DP FP
Scalar:	0.0%	from DP FP
x87 FLOPs:	0.0%	of uOps
Non-FP:	37.7%	of uOps
FP Arith/Mem Rd Instr. Ratio:	6.662	
FP Arith/Mem Wr Instr. Ratio:	495.224	

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
[Loop at line 193 in GSimulation::startompparallel@163]	8.234s	62.9%	100.0%	0.0%	AVX(256); FMA(256)	Body
[Loop at line 165 in GSimulation::startompparallel@163]	0.023s	25.0%	100.0%	0.0%	AVX(128); AVX(256); AVX2(256); FMA(256)	Body

*N/A is applied to non-summable metrics.

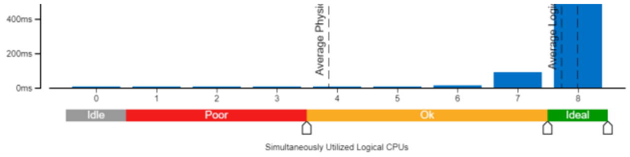
*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

Top Loops/Functions with FPU Usage by CPU Time

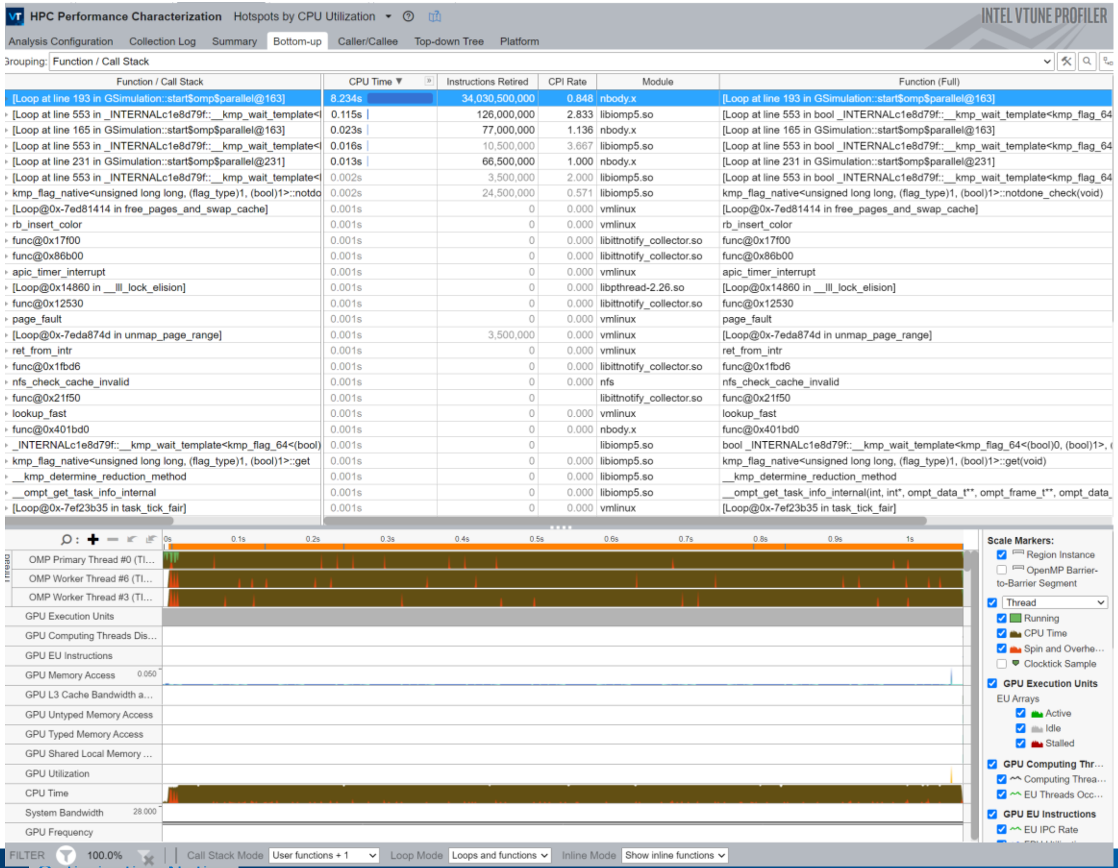
This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops
[Loop at line 193 in GSimulation::startompparallel@163]	8.234s	62.9%	100.0%	
[Loop at line 165 in GSimulation::startompparallel@163]	0.023s	25.0%	100.0%	

*N/A is applied to non-summable metrics.



HPC-Perf: Bottom-up Hotspots view



Function / Call Stack	CPU Time ▼	Instructions Retired	CPI Rate	Module	Function (Full)
[Loop at line 193 in GSimulation::startompparallel@163]	8.234s	34,030,500,000	0.848	nbody.x	[Loop at line 193 in GSimulation::startompparallel@163]
[Loop at line 553 in _INTERNALc1e8d79f::__kmp_wait_template<!	0.115s	126,000,000	2.833	libiomp5.so	[Loop at line 553 in bool _INTERNALc1e8d79f::__kmp_wait_template<kmp_flag_64
[Loop at line 165 in GSimulation::startompparallel@163]	0.023s	77,000,000	1.136	nbody.x	[Loop at line 165 in GSimulation::startompparallel@163]
[Loop at line 553 in _INTERNALc1e8d79f::__kmp_wait_template<!	0.016s	10,500,000	3.667	libiomp5.so	[Loop at line 553 in bool _INTERNALc1e8d79f::__kmp_wait_template<kmp_flag_64
[Loop at line 231 in GSimulation::startompparallel@231]	0.013s	66,500,000	1.000	nbody.x	[Loop at line 231 in GSimulation::startompparallel@231]
[Loop at line 553 in _INTERNALc1e8d79f::__kmp_wait_template<!	0.002s	3,500,000	2.000	libiomp5.so	[Loop at line 553 in bool _INTERNALc1e8d79f::__kmp_wait_template<kmp_flag_64
kmp_flag_native<unsigned long long, (flag_type)1, (bool)1>::notdo	0.002s	24,500,000	0.571	libiomp5.so	kmp_flag_native<unsigned long long, (flag_type)1, (bool)1>::notdone_check(void)
[Loop@0x-7ed81414 in free_pages_and_swap_cache]	0.001s	0	0.000	vmlinux	[Loop@0x-7ed81414 in free_pages_and_swap_cache]
rb_insert_color	0.001s	0	0.000	vmlinux	rb_insert_color
func@0x17f00	0.001s	0	0.000	libittnotify_collector.so	func@0x17f00
func@0x86b00	0.001s	0	0.000	libittnotify_collector.so	func@0x86b00
apic_timer_interrupt	0.001s	0	0.000	vmlinux	apic_timer_interrupt
[Loop@0x14860 in __lll_lock_elision]	0.001s	0	0.000	libpthread-2.26.so	[Loop@0x14860 in __lll_lock_elision]
func@0x12530	0.001s	0	0.000	libittnotify_collector.so	func@0x12530
page_fault	0.001s	0	0.000	vmlinux	page_fault
[Loop@0x-7eda874d in unmap_page_range]	0.001s	3,500,000	0.000	vmlinux	[Loop@0x-7eda874d in unmap_page_range]
ret_from_intr	0.001s	0	0.000	vmlinux	ret_from_intr
func@0x1fbd6	0.001s	0	0.000	libittnotify_collector.so	func@0x1fbd6
nfs_check_cache_invalid	0.001s	0	0.000	nfs	nfs_check_cache_invalid
func@0x21f50	0.001s	0		libittnotify_collector.so	func@0x21f50
lookup_fast	0.001s	0	0.000	vmlinux	lookup_fast
func@0x401bd0	0.001s	0	0.000	nbody.x	func@0x401bd0
_INTERNALc1e8d79f::__kmp_wait_template<kmp_flag_64<(bool)	0.001s	0		libiomp5.so	bool _INTERNALc1e8d79f::__kmp_wait_template<kmp_flag_64<(bool)0, (bool)1>, (
kmp_flag_native<unsigned long long, (flag_type)1, (bool)1>::get	0.001s	0	0.000	libiomp5.so	kmp_flag_native<unsigned long long, (flag_type)1, (bool)1>::get(void)
__kmp_determine_reduction_method	0.001s	0	0.000	libiomp5.so	__kmp_determine_reduction_method
__ompt_get_task_info_internal	0.001s	0	0.000	libiomp5.so	__ompt_get_task_info_internal(int, int*, ompt_data_t**, ompt_frame_t**, ompt_data_
[Loop@0x-7ef23b35 in task_tick_fair]	0.001s	0	0.000	vmlinux	[Loop@0x-7ef23b35 in task_tick_fair]

Use drop down menu to access 'Hotspots by CPU Utilization'

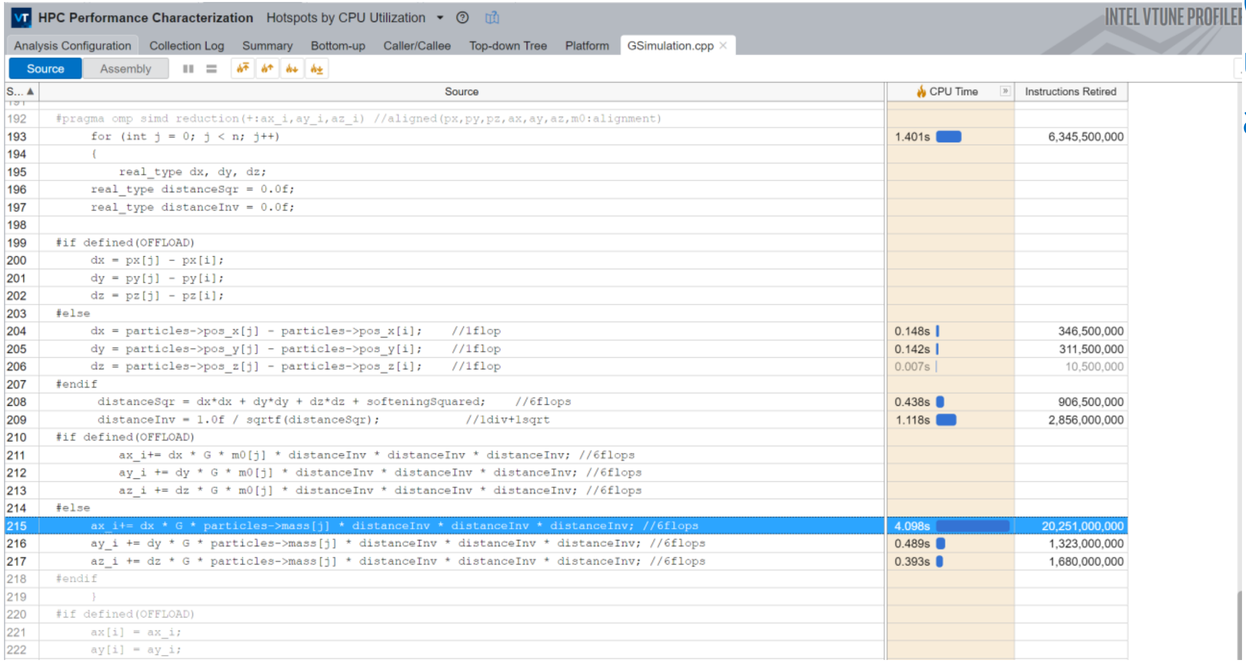
Double click on line to access source and assembly.

Notice the filtering options at the bottom, which allow customization of this view.

Can also do this under "HPC Performance Characterization" and see loop/function data for spin time, serialization, FP Ops, CPI, etc.

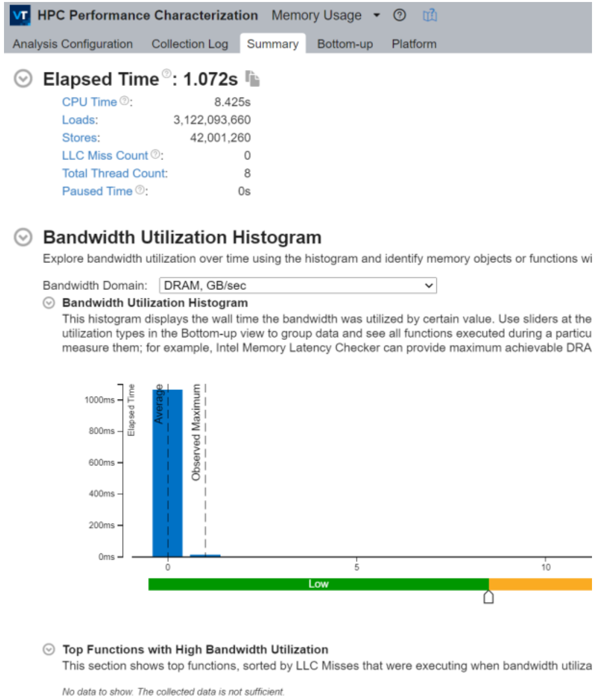
Next steps would include additional analysis to continue the optimization process.

HPC Perf: Bottoms Up – Source View



Click through bottom's up view to see source and metrics.

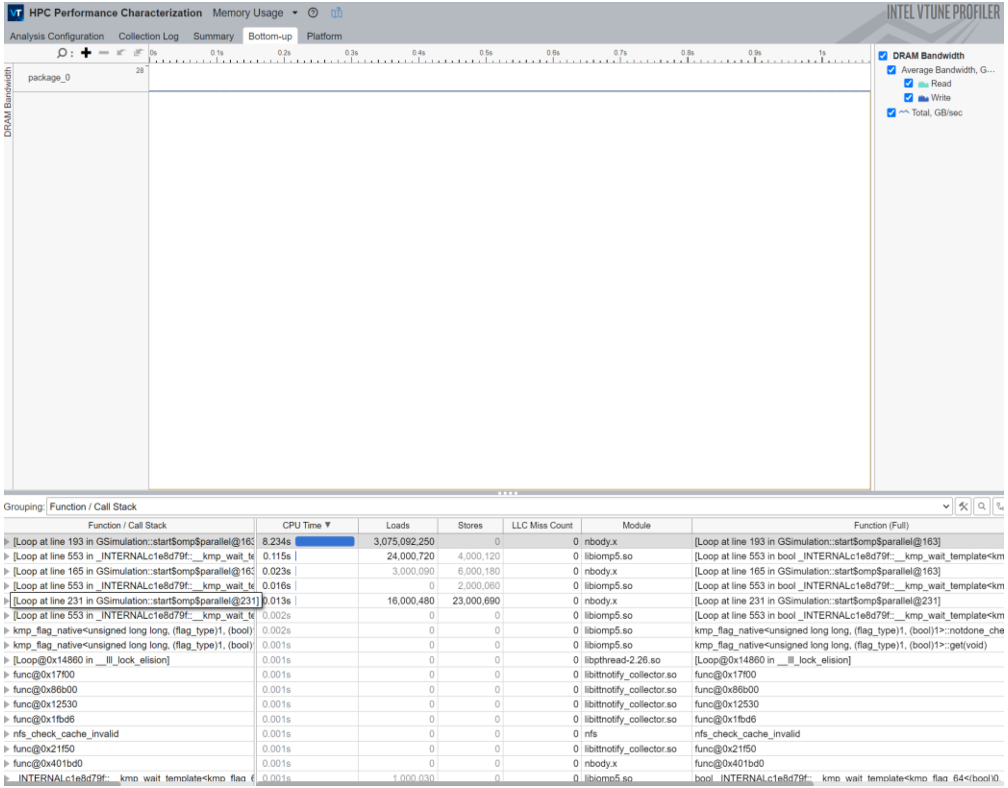
HPC Perf: Memory Usage



Get overview of

- total loads/stores
- bandwidth usage
- L3 bandwidth
- GPU bandwidths
- top functions with high bandwidth utilization.

HPC Perf: Memory Usage, bottom's up view



Can see loads and stores by loop/function

- Sort by loads/stores/llc miss counts

Can also click through as before to see source level view of

- Cpu time
- Loads/stores
- LLC miss counts

HPC-Perf: Hardware Events

HPC Performance Characterization Hardware Events

Analysis Configuration | Collection Log | Summary | Event Count | Sample Count | Uncore Event Count | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 1.072s

CPU Time: 8.425s
CPI Rate: 0.857
Total Thread Count: 8
Paused Time: 0s

Hardware Events

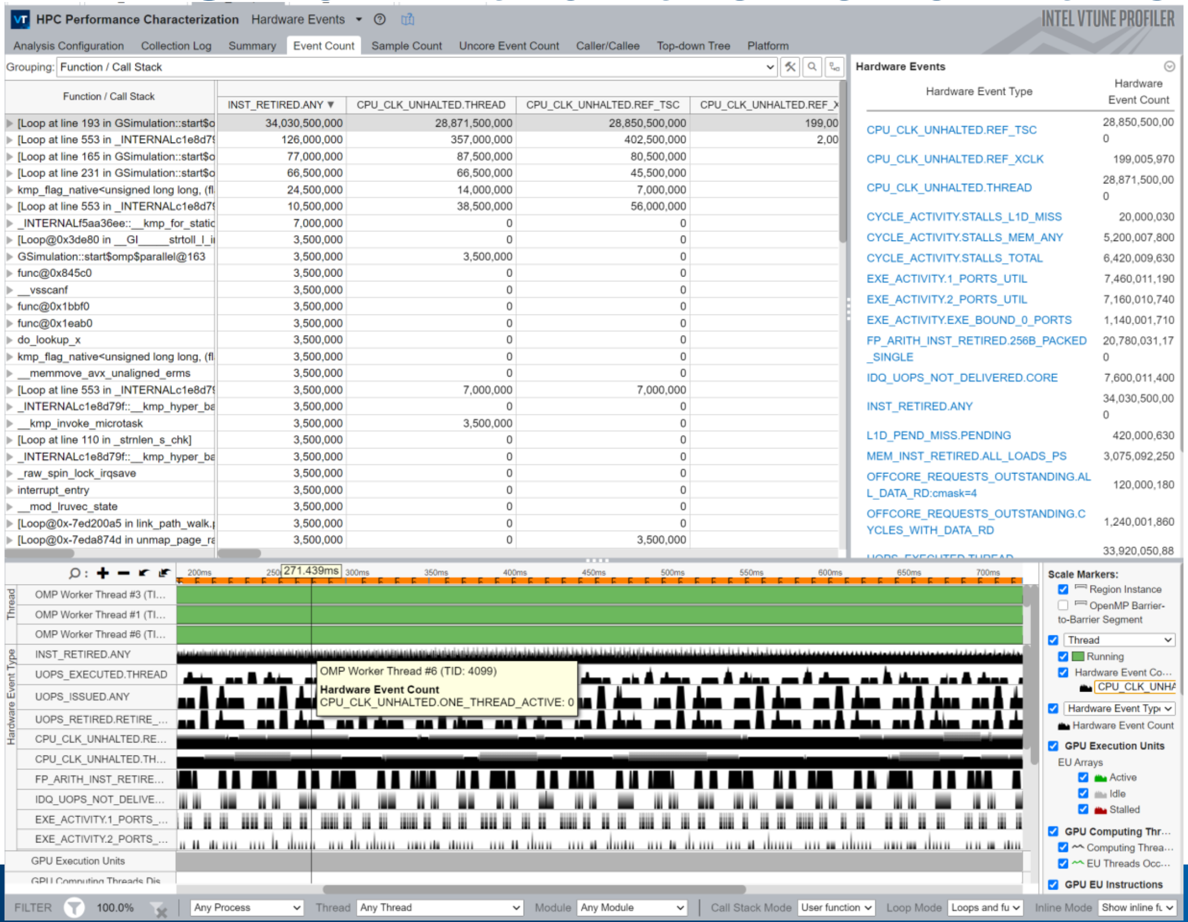
Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	0	0	100003	False
CPU_CLK_UNHALTED.REF_TSC	29,522,500,000	8,435	3500000	False
CPU_CLK_UNHALTED.REF_XCLK	201,006,030	201	100003	False
CPU_CLK_UNHALTED.THREAD	29,505,000,000	8,430	3500000	False
CYCLE_ACTIVITY.STALLS_L1D_MISS	20,000,030	1	2000003	False
CYCLE_ACTIVITY.STALLS_L2_MISS	0	0	2000003	False
CYCLE_ACTIVITY.STALLS_L3_MISS	0	0	2000003	False
CYCLE_ACTIVITY.STALLS_MEM_ANY	5,280,007,920	264	2000003	False
CYCLE_ACTIVITY.STALLS_TOTAL	6,940,010,410	347	2000003	False
EXE_ACTIVITY.1_PORTS_UTIL	7,460,011,190	373	2000003	False
EXE_ACTIVITY.2_PORTS_UTIL	7,180,010,770	359	2000003	False
EXE_ACTIVITY.BOUND_ON_STORES	0	0	2000003	False
EXE_ACTIVITY.EXE_BOUND_0_PORTS	1,140,001,710	57	2000003	False
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	False
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	False
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	False
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	20,800,031,200	1,040	2000003	False
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	2000003	False
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	False
IDQ_UOPS_NOT_DELIVERED.CORE	7,640,011,460	382	2000003	False
INST_RETIRED.ANY	34,415,500,000	9,833	3500000	False
INT_MISC.RECOVERY_CYCLES	0	0	2000003	False
L1D_PEND_MISS.FB_FULL:cmask=1	0	0	2000003	False
L1D_PEND_MISS.PENDING	580,000,870	29	2000003	False
MEM_INST_RETIRED.ALL_LOADS_PS	3,122,093,660	3,122	100003	True
MEM_INST_RETIRED.ALL_STORES_PS	42,001,260	42	100003	True
MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM_PS	0	0	100003	True
MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT_PS	0	0	100003	True
MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS_PS	0	0	100003	True
MEM_LOAD_RETIRED.FB_HIT_PS	0	0	2000003	True
MEM_LOAD_RETIRED.L1_MISS_PS	0	0	2000003	True
MEM_LOAD_RETIRED.L2_HIT_PS	0	0	2000003	True
MEM_LOAD_RETIRED.L3_HIT_PS	0	0	200003	True
MEM_LOAD_RETIRED.L3_MISS_PS	0	0	50003	True
OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD:cmask=4	120,000,180	6	2000003	False
OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DATA_RD	1,240,001,860	62	2000003	False
UOPS_EXECUTED.THREAD	34,300,051,450	1,715	2000003	False
UOPS_EXECUTED.X87	0	0	2000003	False
UOPS_ISSUED.ANY	33,440,050,160	1,672	2000003	False
UOPS_RETIRED.RETIRE_SLOTS	33,400,050,100	1,670	2000003	False

Drop down for Hardware Events.

Summary of all measured performance counters

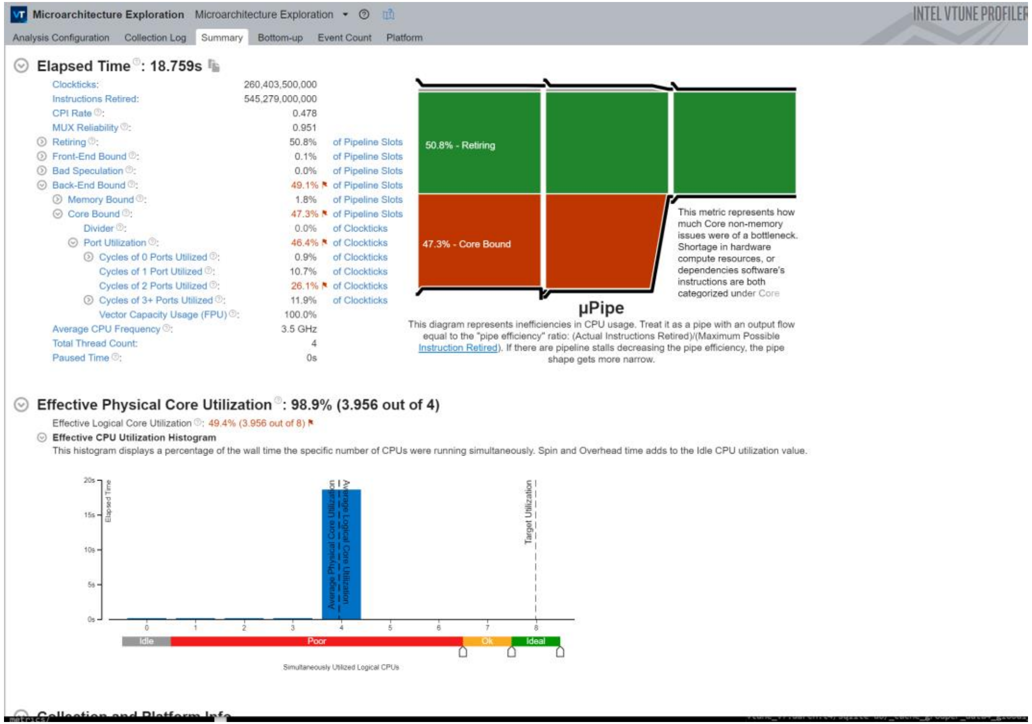
Very similar results to what you'd get from 'uarch-collection'

HPC-Perf: Hardware Performance Counters



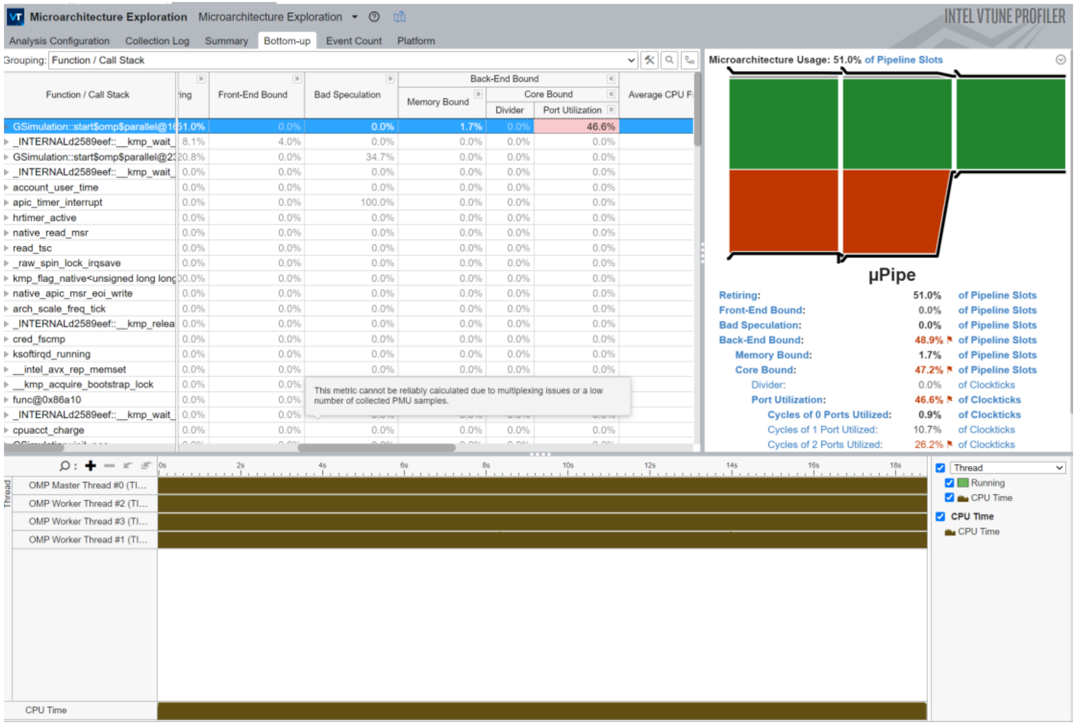
- **Bottom up view of counters**
- Scroll window to see all counters.
- Timeline of counter activity
- Click through to see source Level view of counter data
- CPU thread and GPU counters

uarch-exploration: summary



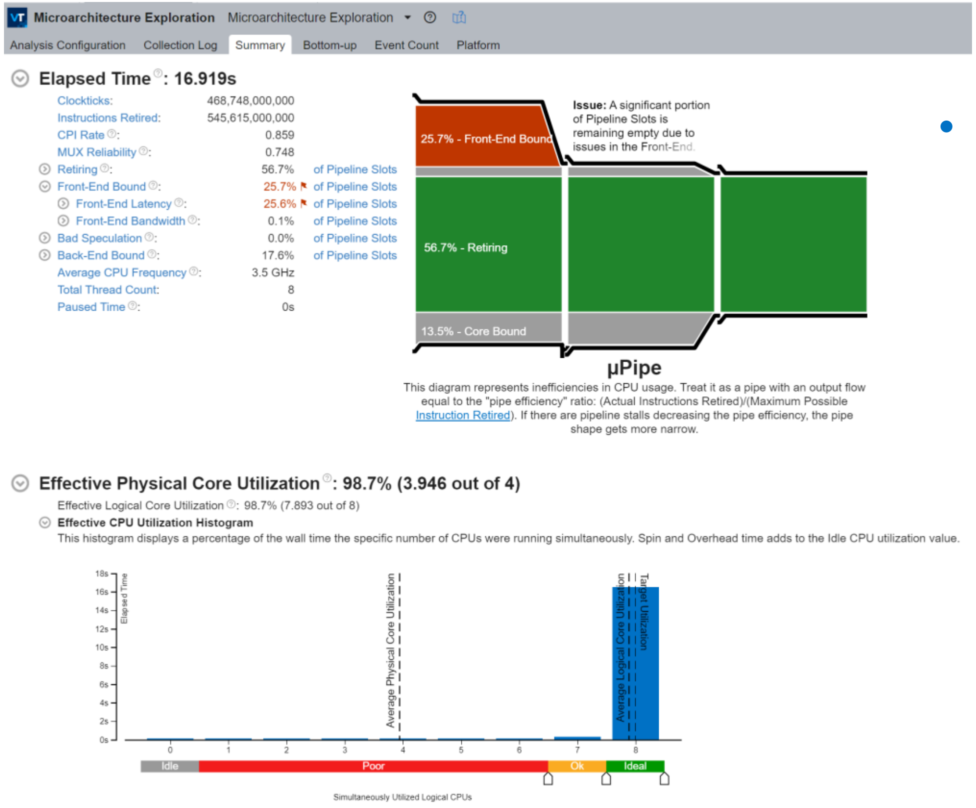
- Running 4 threads
- Want ‘Retiring to be 100%’ – high instruction throughput
- Reporting core bound – implying not enough resources available, in this case likely to be FP units
- Also can look at raw performance counters (including timeline).

uarch-exploration: bottom up



Like other collections, can click through the top hotspots to see source code and where limiter is seen to be sequence of operations with high flop counts.

uarch-exploration: 8 thread summary



- Running with 8 threads improves performance very slightly but shows code is now frontend bound
  - Likely due to pipeline slots being stalled due to too many memory references per cycle.

Memory-Consumption collection:

Memory Consumption Memory Consumption

Analysis Configuration | Collection Log | Summary | Bottom-up | GSimulation.cpp

Elapsed Time: 1.085s

Allocation Size: 8.0 MB
Deallocation Size: 8.0 MB
Allocations: 221
Total Thread Count: 8
Paused Time: 0s

Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
GSimulation::start	7.8 MB	0.0 B	161	nbody.x
_mm_malloc	160.5 KB	0.0 B	11	nbody.x
pool	72.7 KB	0.0 B	1	libstdc++.so.6
GSimulation::GSimulation	4.1 KB	0.0 B	1	nbody.x
_INTERNALa9a94edf::[OpenMP worker]	1.3 KB	0.0 B	28	libiomp5.so
[Others]	528.0 B	0.0 B	19	libgcc_s.so.1

**N/A is applied to non-summable metrics.*

Collection and Platform Info

Shows top memory consumers

Bottom's up show's by loop/function/timeline of consumption

Can click through function/loop to see allocation/deallocation sizes at source level.

VTUNE: ITT

Itt pause & resume

Use __itt_pause() & __itt_resume() to target data collection only in specific regions.

```
#include <ittnotify.h>
  …uninteresting work….
  __itt_resume();
  …interesting work….
  __itt_pause();
  …more uninteresting work…
```

- Launch with `amplxe-cl –start-paused ……`

Using itt to create custom counters

One can create custom counters that show up on VTune timelines by using the itt interface.

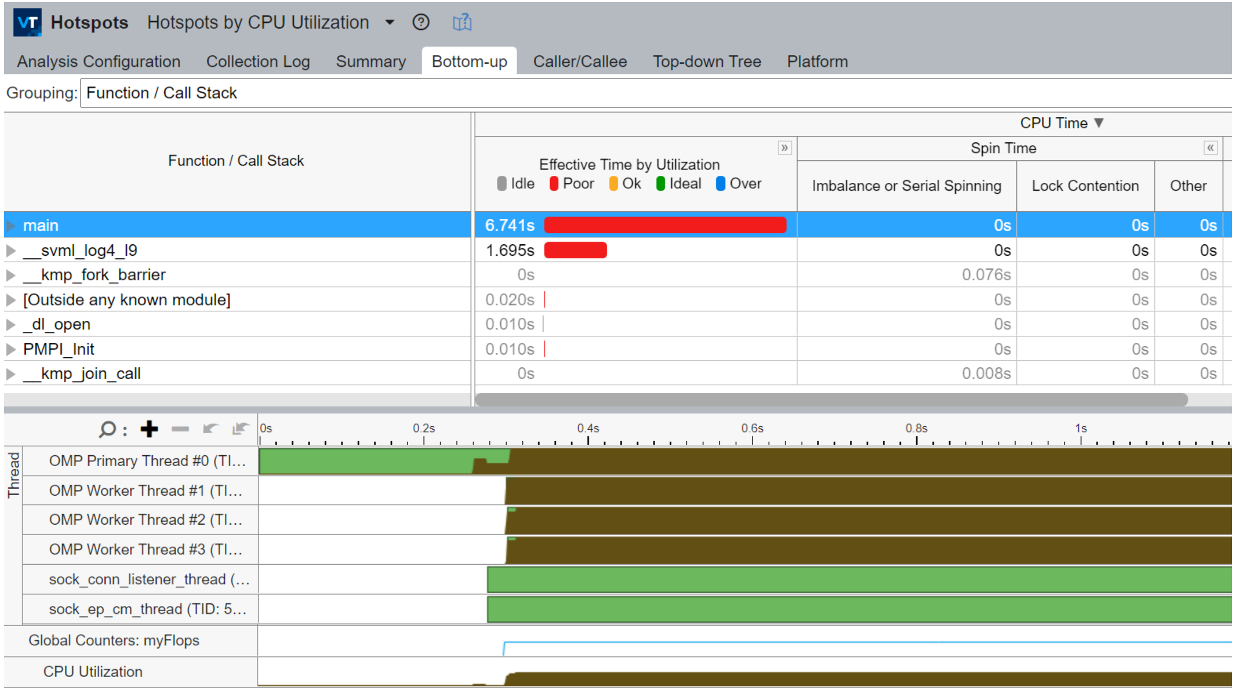
In the example at the right, the counter "myFlops" will show up in the performance metrics timelines.

Tested with hotspot and uarch-exploration and it works. Some issue with hpc-performance that is being looked at.

```
#include <ittnotify.h>

Main()
{

     __itt_counter  myCounter;
     __itt_counter_create(“myFlops”, “Domain”);
     …do some stuff….
     __itt_counter__set_value( myCounter, &val );
     …do some stuff…
     __itt_counter_set_value( myCounter, &val);
     …do more stuff…
     __itt_counter_inc_delta( myCounter, &val );
     …do more stuff….
     __itt_counter_dec_delta( myCounter, &val);

     __itt_counter_destroy(myCounter);

}
```

Counter creation example

Collected using 'hotspots'



VTUNE: TARGETING MPI RANKS

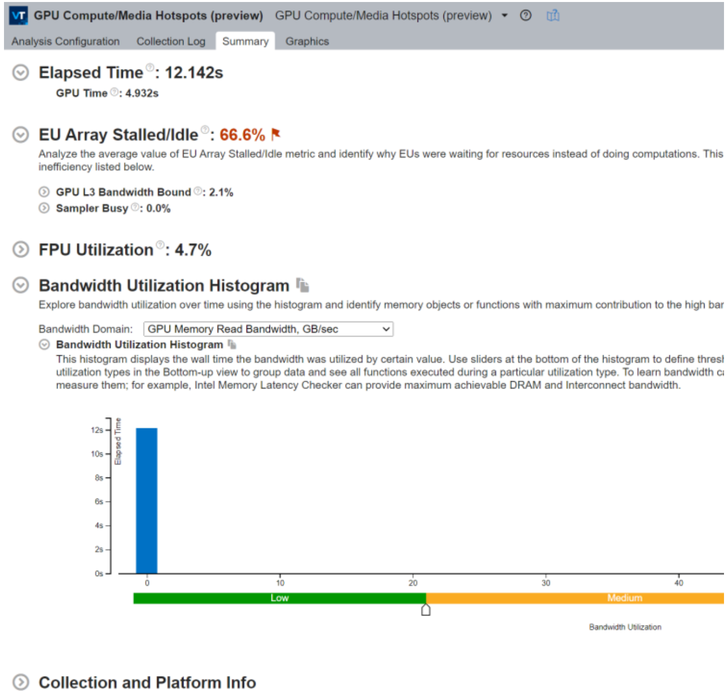
Collecting on Single MPI Ranks

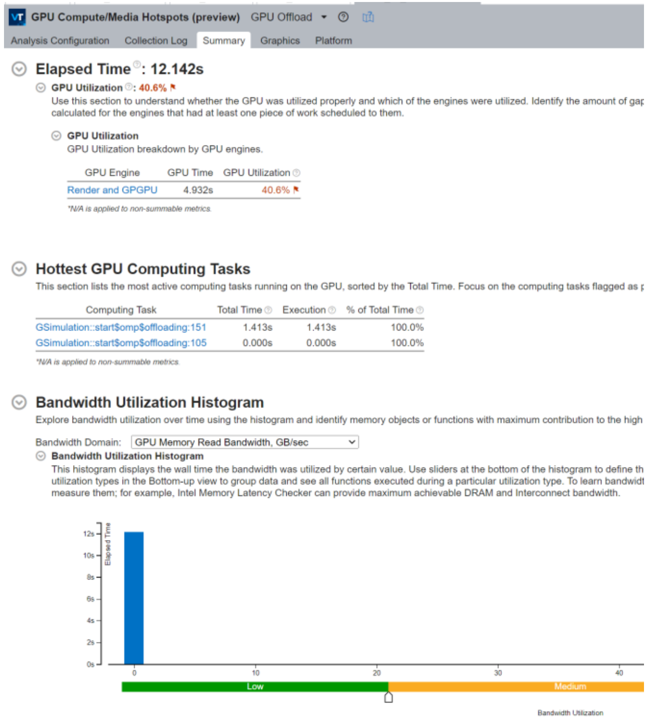
- Might want to use VTune on an MPI application but not, by default, collect data on all MPI ranks as VTune is not designed for that.
  - Still possible to gather some useful data.
  - Using ittnotify is not the route as it still collects data on all ranks even if you pause collection before MPI_Init()
  - Use env vars and MPMD mode:

```
mpirun –genv I_MPI_PIN_PROCESSOR_LIST=0-2,4-7 –n 7 ./app :
       -genv I_MPI_PIN_PROCESSOR_LIST=3 –n 1 amplxe-cl –c hotspots
       –r appdat -- ./app
```

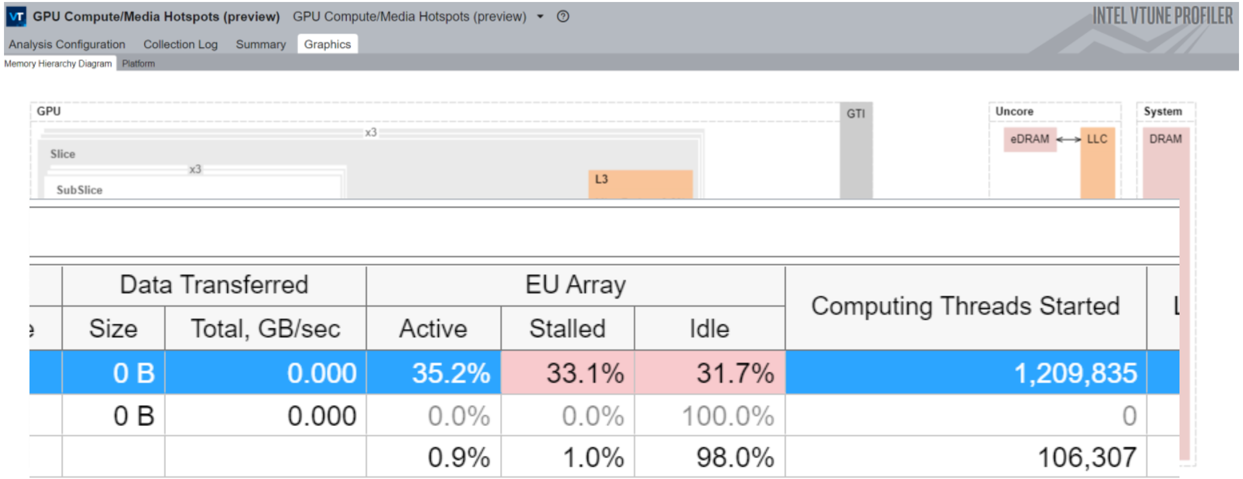
VTUNE: GPU OFFLOADING

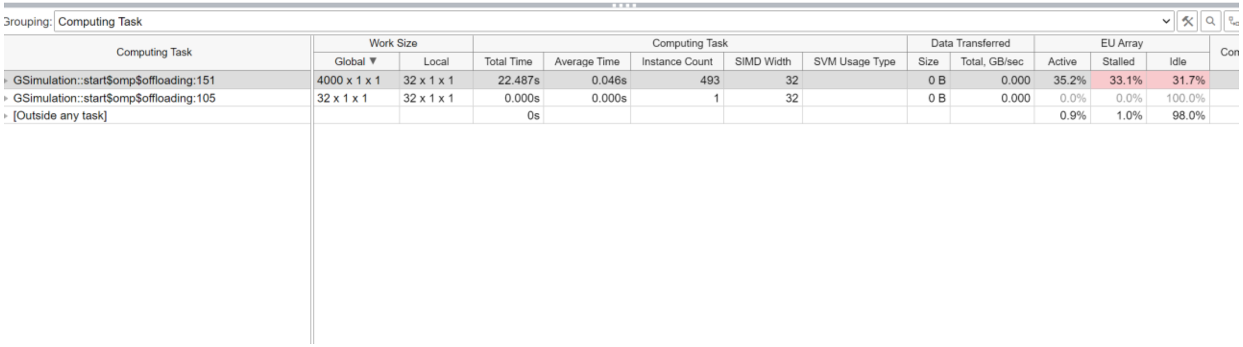
VTune gpu-offloading





- Graphics Information about speeds and feeds (no context)





PROFILING PYTHON & ML APPLICATIONS

Python

Profiling Python is straightforward in VTune™ Amplifier, as long as one does the following:

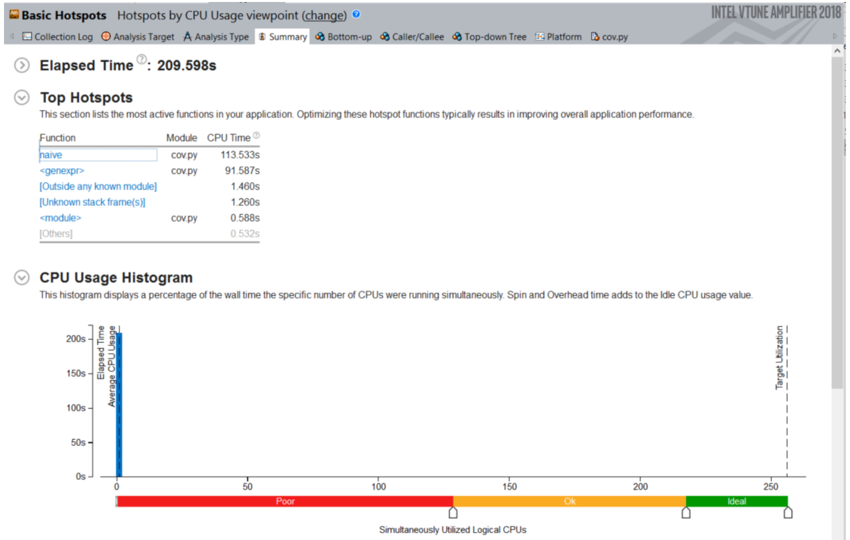
- The “application” should be the full path to the python interpreter used
- The python code should be passed as “arguments” to the “application”

In Theta this would look like this:

```
mpirun -n 1 -N 1 amplxe-cl -c hotspots -r res_dir \
                -- /usr/bin/python3 mycode.py myarguments
```

Simple Python Example on Theta

```
mpirun -n 1 -N 1 amplxe-cl -c hotspots -r vt_pytest \
                          -- /usr/bin/python ./cov.py naive 100 1000
```



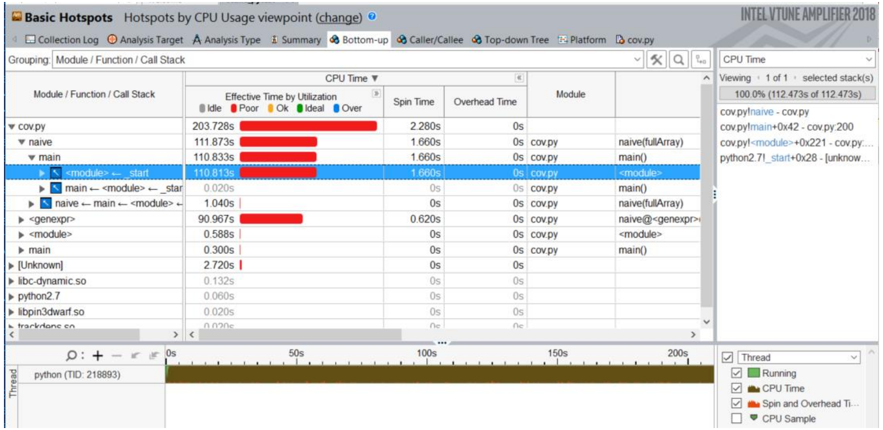
Naïve implementation of the calculation of a covariance matrix

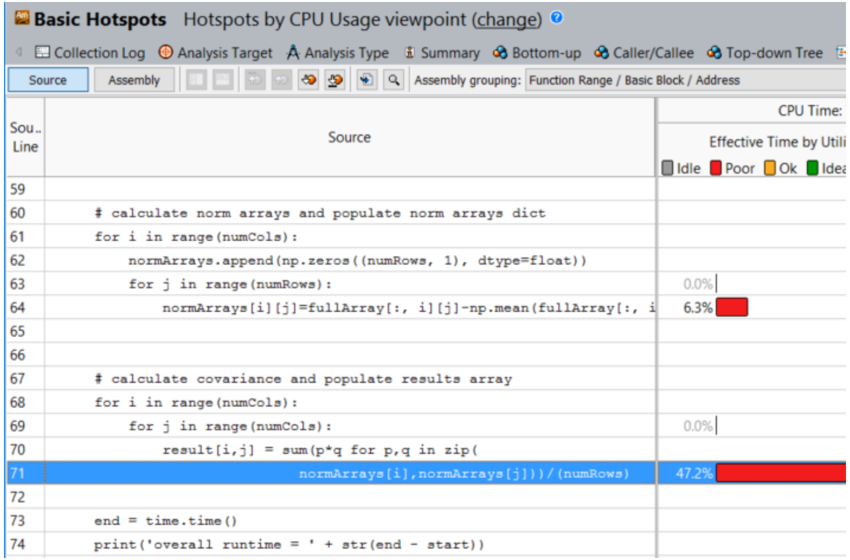
Summary shows:

- Single thread execution
- Top function is “naive”

Click on top function to go to Bottom-up view

Bottom-up View and Source Code





Inefficient array multiplication found quickly

We could use numpy to improve on this

Note that for mixed Python/C code a Top-Down view can often be helpful to drill down into the C kernels

COMMON ISSUES

Fixes

No call stack information/unknown stack frame

- Check finalization log
  - Make sure VTune finds your binary along with libraries that you call

Incompatible database scheme when trying to open result in GUI

- Make sure your local VTune is the same version or newer

VTune sampling driver.. using perf or errors mentioning PMU Resources

- Notify support@alcf.anl.gov or your nearest Intel COE person

TIPS AND TRICKS

Speeding up finalization

Advisor

add `--no-auto-finalize` to the aprun

followed by `advixe-cl R survey …` *without aprun* will cause to finalize on the momnode rather than KNL.

You can also finalize on thetalogin:

cd your_src_dir;

export SRCDIR=`pwd | xargs realpath`

advixe-cl -R survey --search-dir src:=${SRCDIR} ..

VTune

add `--finalization-mode=none` to aprun

followed by `amplxe-cl -R hotspots …` *without aprun* will cause to finalize on momnode rather than KNL

You can also finalize on thetalogin:

cd your_src_dir;

export SRCDIR=`pwd | xargs realpath`

amplxe-cl -R hotspots --search-dir src:=${SRCDIR} ..

Managing overheads

Advisor Dependencies and MAP analyses can have huge overheads

If able, run on reduced problem size. Advisor just needs to figure out the execution flow.

Only analyze loops/functions of interest:

https://software.intel.com/en-us/advixe-cl-user-guide-mark-up-loops

When do I use VTune vs Advisor?

VTune

- What's my cache hit ratio?
- Which loop/function is consuming most time overall? (bottom-up)
- Am I stalling often? IPC?
- Am I keeping all the threads busy?
- Am I hitting remote NUMA?
- When do I maximize my BW?

Advisor

- Which vector ISA am I using?
- Flow of execution (callstacks)
- What is my vectorization efficiency?
- Can I safely force vectorization?
- Inlining? Data type conversions?
- Roofline

BACKUP

VTune Cheat Sheet

Compile with -g -dynamic

amplxe-cl -c hpc-performance -flags -- ./executable

- --result-dir=./amplxe-cl_output_dir
- --search-dir src:=../src --search-dir bin:=./
- -knob enable-stack-collection=true -knob collect-memory-bandwidth=false
- -knob analyze-openmp=true
- -finalization-mode=deferred if finalization is taking too long on KNL
- -data-limit=125 ← in mb
- -trace-mpi for MPI metrics on Theta
- amplxe-cl -help collect survey

https://software.intel.com/en-us/amplxe-cl-amplifier-help-amplxe-cl-command-syntax

Advisor Cheat Sheet

Compile with -g -dynamic

advixe-cl -c roofline/depencies/map -flags -- ./executable

- --project-dir=./advixe_output_dir
- --search-dir src:=../src --search-dir bin:=./
- -no-auto-finalize if finalization is taking too long on KNL
- --interval 1 (sample at 1ms interval, helps for profiling short runs)
- -data-limit=125 ← in mb
- advixe-cl -help

https://software.intel.com/en-us/advixe-cl-help-lin-command-line-interface-reference intel®

Software